UNCLASSIFIED DTIC FILE COPY

(2)

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETEING FORM
1. REPORT NUMBER / 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* 10/13/88 Ada Language Commentaries Volume 2 May 26 1988 - June 23 1988	5. TYPE OF REPORT & PERIOD COVERED May 26 1988 to June 23 1988 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s)	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS Ada Joint Program Office 3D139 (1211 S. Fern, C-107) The Pentagon, Washington DC 20301-3081	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081	12. REPORT DATE September 1988 13. NUMBER OF PAGE 100+
14. MONITORING AGENCY NAME & ADDRESS *(If different from Controlling Office)* AJPO	15. SECURITY CLASS *(of this report)* UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT *(of this Report)*
Approved for public release; distribution unlimited.

DTIC ELECTE OCT 1 9 1988 S D E

AD-A201 024

7. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20. If different from Report)*

UNCLASSIFIED

8. SUPPLEMENTARY NOTES

9. KEYWORDS *(Continue on reverse side if necessary and identify by block number)*
Ada Programming Language; Ada compiler validation capability; Ada standards;Ada commentaries;Ada interpetations;ANSI/MIL-STD-1815A.

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

DD FORM 1473 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

CLEARANCE REQUEST FOR PUBLIC RELEASE OF DEPARTMENT OF DEFENSE INFORMATION

TO: Assistant Secrerary of Defense (*Public Affairs*)

ATTN: Director, Freedom of Information & Security Review, Rm 2C757, Pentagon

SEE INSTRUCTIONS ON REVERSE

(This form is to be used in requesting review and clearance of DoD information proposed for public release in accordance with DoDD 5230.9.)

1. DOCUMENT DESCRIPTION

a. TYPE Report	b. TITLE Ada Commentaries
c. PAGE COUNT 128	d. SUBJECT AREA Ada Programming Language

a. NAME *(Last, First, MI)*	b. RANK	c. TITLE
	d. OFFICE	e. AGENCY

3. PRESENTATION/PUBLICATION DATA

This document contains information concerning the commentaries on the Ada language. The commentaries listed here have been approved by the Ada Board. This information is being made available to the general public so that they may better understand the changes being affected by the commentary process and that they may review those changes.

Keywords: Ada Compiler validation, Standards (KR)

4. POINT OF CONTACT	5. PRIOR COORDINATION
a. NAME (Last, First, MI) CASTOR, Virginia, L. VC	a. OFFICE Ada Joint Program Office
b. TELEPHONE NUMBER (Include Area Code) (202) 694-0210	b. AGENCY R&AT

6. REMARKS

This document has been reviewed within the Ada Joint Program Office. It contains no sensitive or classified information.

TO

884440

7. RECOMMENDATION OF SUBMITTING OFFICE/AGENCY

a. The attached material has Department/Office/Agency approval for public realease (qualifications if any, are indicated in Remarks Section) and clearance for open publication is recommended under provisions of DoDD 5230 9 I am authorized to make this recommendation for release on behalf of:

Director, AJPO

b. Clearance is requested by 880909 *(YYMMDD)*.

c. NAME *(Last, First, MI)*	d. TITLE	e. SIGNATURE [illegible]
f. OFFICE OUSD(A)	g. AGENCY	h. DATE *(YYMMDD)* 880916

DD FORM 1910
82 MAR

PREVIOUS EDITION IS OBSOLETE.

ADA LANGUAGE COMMENTARIES

VOLUME II

26 MAY 1988 - 23 JUNE 1988

3

INDEX

Ada Commentary ai-00012-bi.wj downloaded on Tue Aug 9 17:00:45 EDT 1988

```
!standard 08.03    (17)                                  88-05-23  AI-00012/06
!standard 12.03    (22)
!class binding interpretation 86-10-14
!status approved by WG9/AJPO 88-02-05
!status approved by Director, AJPO 88-02-05
!status approved by WG9/Ada Board 87-12-07
!status approved by Ada Board 87-07-30
!status panel/committee-approved 87-02-17 (reviewed)
!status panel/committee-approved (9-0-1) 86-11-13 (pending editorial review)
!status work-item 83-10-10
!references AI-00398, 83-00139, 83-00815
!topic Declaration of homographs by derivation and instantiation
```

!summary 86-12-15

Derived subprograms can be homographs, and so can subprograms declared in an instance.

!question 86-12-15

8.3(17) says:

> Two declarations that occur immediately within the same declarative region must not be homographs, unless either or both of the following requirements are met: (a) exactly one of them is the implicit declaration of a predefined operation; (b) exactly one of them is the implicit declaration of a derived subprogram. In such cases, a predefined operation is always hidden by the other homograph; a derived subprogram hides a predefined operation, but is hidden by any other homograph. Where hidden in this manner, an implicit declaration is hidden within the entire scope of the other declaration (regardless of which declaration occurs first); the implicit declaration is visible neither by selection nor directly.

Consider the following example (Example 1):

```
package P1 is
    type T is private;
    type S is range 1..10;
    procedure Q (X : S; Y : S);     -- Q1
    procedure Q (X : T; Y : S);     -- Q2
    procedure Q (X : T; Y : T);     -- Q3
private
    type T is new S;     -- derives Q1 and Q2 as homographs,
                         -- but the derived Q1 and Q2 are hidden
                         -- by Q3's explicit declaration
end P1;
```

The full declaration of T declares two derived subprograms, Q1 and Q2. Since these subprograms are homographs, condition (b) of 8.3(17) is violated, so the derived type declaration is illegal. Is this correct?

Now consider a generic unit and an instantiation (Example 2):

```
        generic
            type T1 is private;
            type T2 is private;
        package GP2 is
            procedure PROC (X : T1);
            procedure PROC (Y : T2);
        end GP2;

        package P2 is new GP2 (INTEGER, INTEGER);
```

Is this instantiation legal even though the two declarations of PROC within P2 are homographs and it is not the case that exactly one of these declarations is the implicit declaration of a predefined operation or derived subprogram? The note in 12.3(22) asserts that the instantiation is legal, even though P2 appears to violate 8.3(17).

Now consider a similar example (Example 3):

```
        generic
            type T1 is private;
            type T2 is private;
        package GP3 is
            type T is range 1..10;
            procedure PROC (X : T1; Y : T);
            procedure PROC (Z : T2; Y : T);
        end GP3;

        package P3 is new GP3 (INTEGER, INTEGER);

        type NT is new P3.T;        -- legal?
```

Assuming that GP3's instantiation is legal, the derived type declaration for NT implicitly derives two homographs for PROC. Since the implicit declaration of these homographs appears to violate 8.3(17), is this declaration illegal?

Now consider a similar example (Example 4):

```
        generic
            type T4 is private;
        package GP4 is
            type NT4 is new T4;
        end GP4;

        package P4 is new GP4(P3.T);
```

According to AI-00398, the instance P4 contains implicit declarations of subprograms that are derivable for type P3.T. Since there are two such subprograms and they are homographs, the instance contains declarations of two implicitly declared homographs, seemingly in contradiction with 8.3(17). Is P4 a legal instance? (Note that if it is legal, one of the homographs can be called by writing P4.PROC(X => ...).)

Example 4 may suggest that 8.3(17) is not intended to apply within generic instances, but consider this example (Example 5):

```
        generic
            type T5 is private;
        package GP5 is
            type ARR is array (NATURAL range <>) of T5;
            function "and" (L, R : ARR) return ARR;
        end GP5;

        package P5 is new GP5 (BOOLEAN);
```

In accordance with AI-00398, P5 contains an implicit declaration of the predefined "and" operator for the type P5.ARR. Is this declaration hidden in accordance with 8.3(17) by the user-provided declaration of P5."and"?

In short, when does the rule given in 8.3(17) actually apply?

!recommendation 87-01-18

Two declarations that occur immediately within the same declarative region must not be homographs, unless one or more of the following requirements are met: a) exactly one of them is the implicit declaration of a predefined operation; b) one (or both) of them is the implicit declaration of a derived subprogram; or c) the declarations occur within an instance of a generic unit. In such cases, a predefined operation is always hidden by the other homographs; a derived subprogram hides a predefined operation, but is hidden by any other homographs except a derived subprogram. Where hidden in this manner, an implicit declaration (of a predefined operation or derived subprogram) is hidden within the entire scope of the other declaration (regardless of which declaration occurs first); the implicit declaration is visible neither by selection nor directly.

| !discussion 87-12-07

The note in 12.3(22) shows that it was intended to allow homographs within a generic instance, i.e., 8.3(17)'s restrictions on homographs were not intended to apply within such declarative regions. Example 4 (in conjunction with AI-00398) shows that derived subprograms in an instance can be homographs. This example shows it is reasonable to allow the implicit declaration of derived subprogram homographs, at least within generic instances. If derived subprogram homographs are to be allowed within generic instances, however, it seems reasonable to allow them for any derived type declaration as well; forbidding such homographs outside of generic instances would only create work for implementers, and would have no benefit to programmers. The recommendation reflects these conclusions by allowing derived subprograms to be homographs and by explicitly allowing homographs to be created within generic instances.

The recommendation makes all the examples legal. In addition, for Example 5,
| the predefined operator is hidden.

Ada Commentary ai-00099-ra.wj downloaded on Tue Aug 9 16:58:22 EDT 1988

```
| !standard 13.02    (12)                              88-05-23  AI-00099/12
  !class ramification 85-01-31
| !status approved by WG9/AJPO 88-02-05
| !status approved by Director, AJPO 88-02-05
| !status approved by WG9/Ada Board 87-12-07
| !status approved by Ada Board 87-07-30
  !status panel/committee-approved 87-03-16 (reviewed)
  !status panel/committee-approved (5-0-1) 86-11-14 (pending editorial review)
  !status work-item 86-05-09
  !status returned to committee by WG9 86-05-09
  !status committee-approved (7-0-0) 86-02-21
  !status work-item 85-01-31
  !status received 84-01-10
  !references AI-00138, AI-00341, 83-00237, 83-00651, 83-00705, 83-00738,
          83-00757, 83-00783, 83-00794
  !topic 'SMALL can be specified for a derived fixed point type
```

!summary 87-03-13

A representation clause specifying SMALL for a derived fixed point type is allowed if the resulting model numbers are (representable) values of the parent type and the value specified for SMALL is not greater than the delta of the derived type.

!question 85-01-31

13.2(12) imposes no restriction on the application of 'SMALL to a derived fixed point type. For example, could 0.1, 0.5, 1.0, 2.0, and 4.0 all be specified as values of DF1'SMALL in the example below?

```
type F1 is delta 1.0 range -15.0 .. 15.0;    -- F1'SMALL = 1.0
type DF1 is new F1 delta 4.0;
for DF1'SMALL use ...;
```

| !response 87-12-07

3.4(4) says the set of values of a derived type is a copy of the set of values for the parent type. 3.5.6(3) says an implementation of a real type must include the model numbers of the type and represent them exactly. The effect of specifying SMALL for a fixed point type is to help establish the model numbers of the type. Since the model numbers must be representable values of the type, and since the values of a derived type are determined by the parent type, no representation clause is allowed for a derived fixed point type unless the model numbers determined by the clause are representable values. (In addition, 13.2(12) requires that the specified value of SMALL not exceed the delta of the type.)

With respect to the example, DF1'DELTA is 4.0, so a value specified for DF1'SMALL must not exceed 4.0. In addition, the values of DF1 include at least the model numbers of parent subtype F1. These model numbers are -15.0, -14.0, ..., 14.0, 15.0. If DF1'SMALL is specified to be 4.0, the mantissa for DF1 must be 2, so the model numbers for DF1 will be -12.0, -8.0, -4.0,
| 0.0, 4.0, 8.0, and 12.0. Since these are all values of type F1, such a specification of SMALL is allowed. If the specified value of SMALL is 3.0,

the mantissa for DF1 must be 3 (to ensure the bounds of the subtype are within SMALL of model numbers), so the model numbers for DF1 will be -21.0, -18.0, ..., 18.0, 21.0. If the chosen base type for F1 includes these values (see AI-00341), then the representation clause is allowed. On the other hand, if the base type for F1 has only four mantissa bits, then the range of representable values is just -15.0 .. 15.0, and the representation clause for SMALL would not be allowed.

If the representation clause for DF1'SMALL specifies 0.1, then the mantissa of DF1 must be 8 and the model numbers for DF1 will be -25.5, -25.4, ..., 25.4, 25.5. If these model numbers are not represented exactly in the value set for F1's base type, such a representation clause must be rejected.

These arguments are not affected by the presence of an explicit (or implicit) representation clause for the parent type (see AI-00138).

Ada Commentary ai-00113-bi.wj downloaded on Fri Aug 12 09:50:04 EDT 1988

```
!standard 10.05    (02)                              88-05-23  AI-00113/12
!class binding interpretation 83-11-07
!status approved by WG9/AJPO 88-02-05
!status approved by Director, AJPO 88-02-05
!status approved by WG9/Ada Board 87-12-07
!status approved by Ada Board 87-07-30
!status panel/committee-approved 87-01-19 (reviewed)
!status panel/committee-approved (6-0-0) 86-11-14 (pending editorial review)
!status work-item 86-05-20
!status referred back to committee by WG9 86-05-09
!status committee-approved (9-1-2) 86-02-20
!status committee-approved (7-2-0) 85-11-20 (pending letter ballot)
!status failed letter ballot (6-4-2) 85-11
!status committee-approved (9-1-0) 85-09-04 (subject to letter ballot)
!status work-item 85-04-08
!status received 83-11-07
!references AI-00418, 83-00165, 83-00683, 83-00685, 83-00695, 83-00923
!topic A subunit's with clause can name its ancestor library unit
```

!summary 85-04-08

A with clause for a subunit can name the subunit's ancestor library unit.

!question 85-04-08

10.5(2) says, "A library unit mentioned by the context clause of a subunit must be elaborated before the body of the ancestor library unit of the subunit." Suppose the context clause names the ancestor library unit, e.g.,

```
procedure M is
     procedure SUBUNIT is separate;
begin ... end M;

with M;          -- ancestor library unit is named
separate (M)
procedure SUBUNIT is
begin ... end SUBUNIT;
```

Since M is both a library unit and the ancestor library unit for SUBUNIT, the 10.5(2) wording requires that M be elaborated before itself. Does this mean the context clause in this example is illegal?

!recommendation 85-04-08

A library unit mentioned by the context clause of a subunit must be elaborated no later than the body of the subunit's ancestor library unit.

!discussion 86-09-28

The example given in the question shows that the Standard requires an elaboration that cannot be performed, namely, M must be elaborated before M is elaborated. This does not mean the context clause in the example is illegal; it just means an unimplementable interpretation is specified by the Standard. This difficulty must be resolved, either by stating that such examples are illegal or by giving the example a satisfactory meaning (by reinterpreting 10.5(2)).

Before considering the example given in the question, let's consider a very similar example which is definitely legal. This example is just like the one in the question except the library unit is specified with a declaration instead of a body, so the body containing the body stub is a secondary unit, not a library unit:

```
procedure P1;           -- library unit P1

procedure P1 is         -- secondary unit
    procedure P_SEP is separate;
begin ... end P1;

with P1;
separate (P1)
procedure P_SEP is
begin ... end P_SEP;
```

The legality of this example depends, in part, on 8.6(2), which says:

> The package STANDARD forms a declarative region which encloses every library unit and consequently the main program; the declaration of every library unit is assumed to occur immediately within this package. The implicit declarations of library units are assumed to be ordered in such a way that the scope of a given library unit includes any compilation unit that mentions the given library unit in a with clause.

The with clause for the subunit requires the presence of a library unit named P1 whose scope (in STANDARD) includes the unit being compiled, P_SEP. Since such a library unit exists and can be placed prior to the unit being compiled, the requirements of 8.6(2) are satisfied. In addition, 10.1.1(5) says library unit P1 is directly visible within P_SEP, except where hidden. (Of course, library unit P1 would have been visible in exactly this way even if the with clause had been omitted, since P_SEP is a subunit of P1; 10.2(6) says:

> Visibility within the proper body of a subunit is the visibility that would be obtained at the place of the corresponding body stub (within the parent unit) if the with clauses and use clauses of the subunit were appended to the context clause of the parent unit.

This just gives the rule for determining the visibility of identifiers within P_SEP. In particular, it does not change the visibility that P_SEP already has of library unit P1 and P1's declarations.)

Finally, 10.5(2) says:

> A library unit mentioned by the context clause of a subunit must be elaborated before the body of the ancestor library unit of the subunit.

In the present case, this means the declaration of library unit P1 must be elaborated before the subunit is elaborated, and there is no difficulty in doing so.

In short, for the above example, the redundant with clause causes no problems. Moreover, if P1 were a package or generic unit, the same reasoning would hold.

Now consider the example given in the question, which is like P1 except the parent unit for P_SEP is itself a library unit:

```
procedure M is          -- library unit
    procedure SUBUNIT is separate;
begin ... end M;

with M;
separate (M)
procedure SUBUNIT is
begin ... end SUBUNIT;
```

How does the analysis for this example differ from that for P1? The with clause for SUBUNIT still requires the presence of a library unit named M, and such a library unit exists -- it is the body named M. This body can be placed prior to the subunit (to ensure the scope of the library unit includes SUBUNIT), so this requirement of 8.6(2) can be satisfied. The visibility of library unit M is not changed by the presence (or absence) of the with clause, just as before. Of course, if the with clause naming M were literally appended to the (null) context clause of M, the compilation of M would not be successful because M is a library unit (see AI-00418). But this fact is irrelevant because M is not, in fact, being compiled. The appending of with clauses is just a way of explaining to what extent the with clause augments the visibility the subunit has of library unit M. In this case, appending the with clause gives no additional visibility of library unit M.

Finally, 10.5(2) requires that the library unit mentioned in the context clause (i.e., library unit M) be elaborated before the body of SUBUNIT's ancestor library unit (which in this case, is the body for M, and in this case, the body of the ancestor library unit is also a library unit.) This elaboration rule cannot be obeyed, since a unit cannot be elaborated before it is elaborated.

There are two ways to resolve this problem: consider such examples illegal, or consider them legal and provide a semantic interpretation by saying the library unit mentioned in the context clause of a subunit must be elaborated "no later than" the body of the subunit's ancestor library unit body. This interpretation would mean library unit M must be elaborated no later than when it is elaborated (an easy condition to satisfy!).

It does not seem reasonable to consider the M example illegal when the very similar-seeming P1 example is clearly legal and since similar examples using packages and generic units are legal (see AI-00418). Since the proposed semantic interpretation leads to no contradictions, since no other difficulties are introduced by allowing such redundant with clauses, and since all validated compilers allow such examples, it is reasonable to allow the with clause of a subunit to name its ancestor library unit and to require that the ancestor library unit be elaborated no later than the body of the ancestor library unit.

Ada Commentary ai-00146-ra.wj downloaded on Fri Aug 12 09:50:02 EDT 1988

```
!standard 03.05.09 (16)                                88-05-23  AI-00146/10
!standard 03.05.09 (14)
!class ramification 86-11-14
!status approved by WG9/AJPO 88-02-05
!status approved by Director, AJPO 88-02-05
!status approved by WG9/Ada Board 87-12-07
!status approved by Ada Board 87-07-30
!status panel/committee-approved 87-03-16 (reviewed)
!status panel/committee-approved (6-0-0) 86-11-14 (pending editorial review)
!status work-item 86-05-20
!status referred back to committee by WG9 86-05-09
!status committee-approved (8-1-0) 85-11-20
!status work-item 85-04-08
!status received 84-01-10
!references 83-00238, 83-00738, 83-00805
!topic Model numbers for a fixed point subtype with length clause
```

!summary 85-04-08

If a length clause specifying SMALL has been given for a fixed point type, T, then the value of SMALL for any subtype of T is given by T'SMALL.

!question 86-12-17

3.5.9(16), which is a note, says:

> If S is a subtype of a fixed point type or subtype T, then the set of model numbers of S is a subset of those of T. If a length clause has been given for T, then both S and T have the same value for SMALL.

3.5.9(14) defines how model numbers are defined for a fixed point subtype:

> [The elaboration of a fixed point subtype indication] creates a fixed point subtype whose model numbers are defined by the corresponding fixed point constraint and also by the length clause specifying small, if there is one.

Now consider an example:

```
type F is delta 0.1 range -15.0 .. 15.0;
for F'SMALL use 0.1;
subtype FS is F delta 0.8;
```

There is no length clause specified for subtype FS and none is allowed. What is the value of FS'SMALL?

!response 87-03-13

Since a length clause cannot be given for a subtype declared by a subtype declaration (see 13.2(3)), when 3.5.9(14) mentions "the" length clause specifying SMALL, it can only be referring to a length clause given for a declared type. 3.5.9(5) then defines how such a length clause determines the model numbers and the value of SMALL:

> For the model numbers defined by a fixed point constraint, the number SMALL is chosen as the largest power of two that is not greater than the delta of the fixed accuracy definition. Alternatively, it is possible to specify the value of SMALL by a length clause, in which case model numbers are multiples of the specified value.

Consequently, if a length clause is given for the declared type, F, the clause defines the value of SMALL for all subtypes of F, in accordance with 3.5.9(5).

Although the value of SMALL is fixed by a length clause, the length of the mantissa, and hence, the set of model numbers, can change for a subtype:

```
subtype FS3 is F delta 0.8 range -7.0 .. 7.0;
```

F'MANTISSA = FS'MANTISSA, but since FS3 has half the range of F, FS3'MANTISSA = F'MANTISSA - 1.

Ada Commentary ai-00158-bi.wj downloaded on Fri Aug 12 09:49:41 EDT 1988

```
| !standard 10.05    (01)                              88-05-23  AI-00158/05
  !class binding interpretation 87-01-15
| !status approved by WG9/AJPO 88-02-05
| !status approved by Director, AJPO 88-02-05
| !status approved by WG9/Ada Board 87-12-07
  !status approved by Ada Board 87-07-30
  !status panel/committee-approved 87-05-06 (reviewed)
  !status panel/committee-approved (9-0-0) 87-02-17 (pending editorial review)
  !status work-item 87-01-15
  !status received 84-01-10
  !references AI-00222, 83-00226
  !topic The main program is elaborated before it is called
```

!summary 87-01-15

The main program is elaborated before it is called.

!question 87-01-15

Paragraph 10.5(1) lists the library units elaborated before a main program is called; this set does not include the body of the main program itself. Paragraph 10.1(8) states, "Each main program acts as if called by some environment task...." Thus a main program is called before its body is elaborated. According to section 3.9, this raises PROGRAM_ERROR. Is this the intent?

!recommendation 87-01-15

The main program is elaborated before it is called by the environment task.

!discussion 87-01-17

It was clearly not the intent for a call of the main program to raise PROGRAM_ERROR. 10.5 should have mentioned that the body of the main program is elaborated before it is called.

Ada Commentary ai-00170-bi.wj downloaded on Tue Aug 9 14:27:04 EDT 1988

```
| !standard 08.05    (05)                                88-06-13  AI-00170/07
  !class binding interpretation 84-01-17
| !status approved by WG9/AJPO 88-02-05 (corrected in accordance with AI-00502)
| !status approved by WG9/AJPO 86-07-22
  !status approved by Director, AJPO 86-07-22
  !status approved by WG9/Ada Board 86-07-22
  !status approved by Ada Board 86-07-22
  !status approved by WG9 86-05-09
  !status committee-approved (8-0-0) 86-02-20
  !status work-item 86-01-17
  !status received 84-01-17
| !references 83-00257, 83-00859
  !topic Renaming a slice
```

!summary 86-01-17

A slice must not be renamed if renaming is prohibited for any of its components.

| !question 88-06-13

4.1.2(1) says that:

> A slice denotes a one-dimensional array formed by a sequence of consecutive components of a one-dimensional array.

and 8.5(5) says that:

> The following restrictions apply to the renaming of a subcomponent that depends on discriminants of a variable. ...

A strict reading of 4.1.2 indicates that a slice is not an example of a component -- it is, instead, a one-dimensional array, which is apparently an entirely separate concept. If so, then apparently the restrictions in 8.5(5) about renaming a subcomponent that depends on a discriminant do not apply to slices. For example:

```
|       type SINT is NATURAL range 0..100;
        type VREC (N : SINT := 0) is
             record
                  S : STRING (1..N);
             end record;

        OBJ : VREC := (3, "ABC");

        OBJ1 : CHARACTER renames OBJ.S(1);       -- illegal by 8.5(5)
        OBJ2 : STRING    renames OBJ.S(1..2);    -- illegal? (yes)
```

OBJ.S(1..2) is a slice and thus not clearly a "component", so 8.5(5) does not clearly apply. Do the restrictions of 8.5(5) apply to slices as well as "just subcomponents"?

!recommendation 86-01-17

A slice must not be renamed if renaming is prohibited for any of its components.

!discussion 86-03-05

The reason for the restrictions in 8.5(5) is to prevent the newly declared name from denoting an object whose existence may subsequently cease while execution is still within the scope of the name. In the example, a subsequent assignment

```
        OBJ := (0, "");
```

would cause OBJ1 to denote a no-longer existing object, namely, OBJ.S(1). Similarly, OBJ2 would now denote a non-existent array object, namely, OBJ.S(1..2). This undesirable situation was not intended.

Ada Commentary ai-00179-ra.wj downloaded on Tue Aug 9 14:27:02 EDT 1988

!standard 03.05.10 (08) 88-06-13 AI-00179/08
!class ramification 85-09-05
!status approved by WG9/AJPO 88-02-05 (corrected in accordance with AI-00467)
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 85-11-18
!status committee-approved (10-0-0) 85-09-05
!status work-item 84-06-11
!status received 84-01-25
!references 83-00265, 83-00654, 83-00803
!topic The definition of the attribute FORE

!summary 85-09-18

The attribute 'FORE is defined in terms of the decimal representation of model numbers.

!question 85-09-18

For a fixed point type definition such as

```
        type F is delta 0.1 range 0.0 .. 9.96;
        for F'SMALL use 0.01;
```

is the value of F'FORE 2 or 3? Note that when outputting F'LAST with an AFT of 1, the string " 10.0" will be produced, and this string requires a FORE of 3.

!response 88-06-13

The Standard gives the following definition for 'FORE:

> Yields the minimum number of characters needed for the integer part of the decimal representation of any value of the subtype T, assuming that the representation does not include an exponent, but includes a one-character prefix that is either a minus sign or a space. ...

For a fixed point subtype declared as follows:

```
        type F is delta 0.1 range 0.0 .. 9.96;
        for F'SMALL use 0.01;
```

the value of 'FORE is 2 since the straightforward interpretation of "decimal representation of any value of the subtype" means the exact decimal representation of the model numbers belonging to F. In this case, the value 2 is unsuitable when certain values, e.g., 9.96, are output with an AFT of 1. It is up to the programmer to take this effect into account when using fixed point output formats.

The value returned by 'FORE can be implementation dependent. For example:

```
        type G is delta 0.01 range 1.00 .. 10.00;
        for G'SMALL use 0.01;
        subtype SG is F delta 0.01 range 1.00 .. 9.995;
```

For the subtype SG, 9.99 and 10.00 are consecutive model numbers (3.5.9(14)). It is implementation dependent whether the upper bound of SG is represented as the model number 9.99 or the model number 10.0. Depending on the implementation's choice, the value returned by SG'FORE will be either 2 or 3. In addition, note that the bounds of SG need not be given by static expressions. If the upper bound is non-static and has a value lying in the model interval 9.99 to 10.00, SG'FORE's value will be implementation dependent (and must be computed at run-time). The fact that 'FORE may return implementation dependent values should be taken into consideration by programmers.

Ada Commentary ai-00192-ra.wj downloaded on Fri Aug 12 09:49:39 EDT 1988

```
| !standard 08.06    (02)                              88-05-23  AI-00192/05
  !class ramification 84-03-13
| !status approved by WG9/AJPO 88-02-05
| !status approved by Director, AJPO 88-02-05
| !status approved by WG9/Ada Board 87-12-07
  !status approved by Ada Board (21-0-0) 87-02-19
  !status panel/committee-approved 86-10-15 (reviewed)
  !status panel/committee-approved (5-0-0) 86-09-11 (pending editorial review)
  !status work-item 86-08-07
  !status received 84-03-13
  !references 83-00300
  !topic Allowed names of library units
```

!summary 86-08-12

The name of a library unit cannot be a homograph of a name that is already declared in package STANDARD.

!question 86-08-07

8.6(2) states:

> The package STANDARD forms a declarative region which encloses every library unit and consequently the main program; the declaration of every library unit is assumed to occur immediately within this package.

Does this imply that a library unit may not have a name such as STRING which is already declared in STANDARD?

!response 86-09-15

Since the name of a library unit is implicitly declared in STANDARD, the name cannot be a homograph (8.3(15)) of a name that is already declared in package STANDARD. In particular, a library unit cannot have the name BOOLEAN, INTEGER, FLOAT, CHARACTER, ASCII, NATURAL, POSITIVE, STRING, DURATION, CONSTRAINT_ERROR, NUMERIC_ERROR, PROGRAM_ERROR, STORAGE_ERROR, or TASKING_ERROR. In addition, if an implementation has provided predefined numeric types such as LONG_INTEGER, SHORT_INTEGER, etc., a library unit cannot have any of these names. Similarly, no library unit package or generic unit can have the name TRUE or FALSE, but a library unit subprogram can have the name TRUE or FALSE as long as it is not a homograph of the enumeration literals TRUE or FALSE (i.e., as long as it is not a parameterless function with return type STANDARD.BOOLEAN).

Ada Commentary ai-00195-ra.wj downloaded on Fri Aug 12 09:49:07 EDT 1988

```
| !standard 09.06    (05)                           88-05-23  AI-00195/09
  !class ramification 84-03-13
| !status approved by WG9/AJPO 88-02-05
| !status approved by Director, AJPO 88-02-05
| !status approved by WG9/Ada Board 87-12-07
  !status approved by Ada Board 87-07-30
  !status panel/committee-approved (9-1-3) 87-02-17 (by ballot)
  !status panel/committee-approved (5-0-0) 86-11-14 (pending letter ballot)
  !status work-item 86-08-07
  !status received 84-03-13
  !references AI-00325, AI-00201, 83-00296, 83-00864
  !topic The intended use of CLOCK
```

!summary 87-08-20

CLOCK returns a value that reflects the time of day in the external environment.

!question 86-12-28

What is the intended function of CLOCK? In particular, must successive calls to CLOCK produce monotonically nondecreasing values?

!response 87-06-11

The Standard only requires that CLOCK return values reflecting the behavior of a hardware clock. Successive calls to CLOCK have properties that depend on the execution environment. For example, if the hardware clock is reset by the system operator (to compensate for a change to Daylight Saving Time, power failures, or inaccurate time-keeping), successive calls to CLOCK can fail to produce monotonically nondecreasing values. Similarly, successive calls to CLOCK could exhibit odd behavior if the environment consists of a set of processors each of which provides its own hardware clock. In any case, failure to produce monotonically nondecreasing values would require justification in terms of AI-00325.

Ada Commentary ai-00198-bi.wj downloaded on Fri Aug 12 09:49:06 EDT 1988

```
| !standard 09.10    (05)                                 88-05-23  AI-00198/09
  !standard 09.03    (04)
  !standard 09.03    (08)
  !class binding interpretation 84-03-13
| !status approved by WG9/AJPO 88-02-05
| !status approved by Director, AJPO 88-02-05
| !status approved by WG9/Ada Board 87-12-07
  !status approved by Ada Board 87-07-30
  !status panel/committee-approved 87-05-06 (reviewed)
  !status panel/committee-approved (9-0-0) 87-02-17 (pending editorial review)
  !status work-item 86-01-21
  !status received 84-03-13
  !references AI-00149, 83-00290, 83-00438, 83-00766, 83-00863
  !topic Termination of unactivated tasks
```

!summary 87-03-13

If a task is abnormally completed, then any task it has created but not yet activated becomes terminated and is never activated.

If PROGRAM_ERROR is raised before attempting to activate one or more tasks because the bcdy of at least one of these tasks has not yet been elaborated (see AI-00149), all the unactivated tasks become terminated.

!question 87-01-20

A task can be aborted while attempting to activate some other tasks. The intention is that unactivated tasks in such a case become terminated, but the existing rules do not seem to cover all the cases that can arise. In particular, suppose the aborted task is executing an allocator and the allocated tasks do not depend on the aborted task:

```
        procedure P is

             task ACTIVATOR;
             task type T;
             function F return INTEGER;

             type R is
                  record
                       C1 : T;
                       C2 : NATURAL := F;  -- aborts self before C1 is activatec
                  end record;
             type A is access R;

             function F return INTEGER is
             begin
                  abort ACTIVATOR; -- abort self (F called within ACTIVATOR)
                  return 3;
             end F;
```

```
          task body T is
          begin
               null;
          end T;

          task body ACTIVATOR is
               X : A := new R;  -- X.C1 depends on P, not on ACTIVATOR
          begin
               null;
          end ACTIVATOR;

     begin  -- activate ACTIVATOR here
          null;
     end P;
```

By 4.8(6), the allocator first creates the object, X.all, and then initializes it (3.2.1(15)). The attempt to initialize X.C2 aborts task ACTIVATOR and causes it to become abnormal. But task X.C1 depends on P, not on ACTIVATOR, and so by 9.10(4), task X.C1 doesn't become abnormal. But by 9.3(6), X.C1 does not begin activation until after X.all is initialized. Thus, X.C1 is created, but is never activated, nor is it ever abnormal, completed, or terminated. Is this correct?

Finally, suppose an attempt is made to activate a task before its body has been elaborated:

```
     declare
          task type PROG_ERR;

          package P is ... end P;

          package body P is
               X : PROG_ERR;
          begin
               -- the attempt to activate X raises PROGRAM_ERROR
               null;
          exception
               when others =>
                    -- X'TERMINATED could be false?
          end P;

          task body PROG_ERR is ... end PROG_ERR;
     begin ... end;
```

The attempt to activate X does not take place during the elaboration of a declarative part; it occurs prior to the execution of the sequence of statements, so 9.3(4) does not apply. Since X is never activated, is it the intent that X be considered terminated?

!recommendation 87-03-13

If a task is abnormally completed, then any task it has created but not yet activated becomes terminated and is never activated.

If PROGRAM_ERROR is raised before attempting to activate one or more tasks because the body of at least one of these tasks has not yet been elaborated, all the unactivated tasks become terminated.

!discussion 87-01-20

9.3(4) says:

> Should an exception be raised by the activation of [a task object declared in a declarative part or package specification, either directly or as a subcomponent of an object], that task becomes a completed task (see 9.4); other tasks are not directly affected.

9.3(8) says:

> Should an exception be raised by the initialization of the object created by an allocator (hence before the start of any activation), any task designated by a subcomponent of this object becomes terminated and is therefore never activated.

With respect to the first example, aborting a task does not raise an exception, so task X.C1 is indeed created and never activated nor terminated. However, 9.3(4, 8) show the intent in this case is that X.C1 be considered terminated.

Similarly, 9.3(4) and 9.3(8) do not cover the case where PROGRAM_ERROR is raised prior to the process of attempting to activate one or more tasks (see AI-00149), but the intent is that if PROGRAM_ERROR is raised for this reason, all of the unactivated tasks are terminated.

Ada Commentary ai-00201-ra.wj downloaded on Thu Aug 11 16:50:53 EDT 1988

```
| !standard 13.07.01 (07)                                88-05-23  AI-00201/07
  !standard 09.06    (05)
  !standard 09.06    (04)
  !standard 09.06    (01)
  !class ramification 84-03-13
| !status approved by WG9/AJPO 88-02-05
| !status approved by Director, AJPO 88-02-05
| !status approved by WG9/Ada Board 87-12-07
  !status approved by Ada Board 87-07-30
  !status panel/committee-approved 87-05-06 (reviewed)
  !status panel/committee-approved (9-0-0) 87-02-17 (pending editorial review)
  !status work-item 86-07-30
  !status received 84-03-13
  !references AI-00195, AI-00325, AI-00366, 83-00293, 83-00321, 83-00774,
              83-00864
  !topic The relation between TICK, CLOCK, and the delay statement
```

!summary 86-12-28

The value returned by successive calls to the CLOCK function can be expected to change at the frequency indicated by SYSTEM.TICK.

There is no required relation between SYSTEM.TICK and DURATION'SMALL.

Delay statements need not be executed with an accuracy that is related to SYSTEM.TICK or DURATION'SMALL; in particular, delay statements can be executed more accurately than SYSTEM.TICK implies. Execution with less accuracy than SYSTEM.TICK requires justification in terms of AI-00325.

!question 86-12-28

Is it the intention that SYSTEM.TICK be a quantification of the accuracy of CALENDAR.CLOCK? If not, what is intended?

In 13.7.1(7), what does "basic clock period" mean? (This is also called "basic clock cycle" at 9.6(4).)

Is there any required relation between SYSTEM.TICK and DURATION'SMALL? In particular, can SYSTEM.TICK be 1.0 second when DURATION'SMALL is 20 ms?

Is a delay statement executed with an accuracy that is related to the value of SYSTEM.TICK or DURATION'SMALL?

!response 87-08-20

The CLOCK function returns values associated with a hardware clock (see AI-00195). The "basic clock period" mentioned in the description of SYSTEM.TICK refers to the frequency with which this clock is updated. For example, suppose the hardware clock is updated twice at 8 millisecond intervals and then once after a 9 millisecond interval. The average update rate is 8 1/3 milliseconds (1/120 second). SYSTEM.TICK should be 1.0/120.0.

There is no required relationship between the value of SYSTEM.TICK and the value of DURATION'SMALL (as is stated in 9.6(4), since the "basic clock cycle" (mentioned in 9.6(4)) and the "basic clock period" (mentioned in 13.7.1(7)) are the same).

The accuracy of the delay imposed by a delay statement is not related directly to the value of SYSTEM.TICK nor to the value of DURATION'SMALL since the value of SYSTEM.TICK only reflects the frequency with which successive calls to CLOCK can be expected to change, and the value of DURATION'SMALL only reflects the accuracy with which values of type DURATION can be represented. Of course, since the clock used for the function CLOCK can also be the clock used to schedule delay statements, it can be expected that in a reasonable implementation, delay statements will be executed with an accuracy that is no worse than SYSTEM.TICK. An accuracy that is significantly worse would require justification in terms of AI-00325.

Ada Commentary ai-00209-ra.wj downloaded on Thu Aug 11 16:50:51 EDT 1988

```
| !standard 04.10    (04)                          88-05-23  AI-00209/06
  !class ramification 84-03-13
| !status approved by WG9/AJPO 88-02-05
| !status approved by Director, AJPO 88-02-05
| !status approved by WG9/Ada Board 87-12-07
  !status approved by Ada Board 87-07-30
  !status panel/committee-approved 87-05-06 (reviewed)
  !status panel/committee-approved (8-0-0) 87-02-18 (pending editorial review)
  !status work-item 87-01-15
  !status received 84-03-13
  !references AI-00325, 83-00284
  !topic Exact evaluation of static universal real expressions
```

!summary 87-06-11

An implementation can refuse to evaluate a static universal real expression only if there are insufficient resources to evaluate the expression exactly, e.g., if there is insufficient memory available. Inexact results must not be delivered.

!question 84-03-13

Consider the following static expressions:

```
(a)   (1.0/3.0) * 3.0 = 1.0
(b)   1.0E-1000 + 1.0
(c)   (1.0E1000 + 1.0) / 1.0E1000 = 1.0
```

A typical floating point evaluation of expression (a) would yield the value FALSE, but 4.10(4) requires that the evaluation of static universal real expressions be exact. Hence, in order to evaluate universal real expressions exactly, a compiler must include a rational arithmetic package or some even more complicated expression representation and manipulation package. Such a rational arithmetic package would consume more than 3000 bits to represent the exact value of expression (b). Any limitation on the precision of the arithmetic would result in expression (c) evaluating to TRUE, which would be incorrect.

Can an implementer employ a limited precision evaluation strategy for static universal real expressions, rejecting programs that cannot be evaluated exactly using this strategy?

!response 87-03-24

The requirement to evaluate static universal real expressions exactly was given careful consideration during Ada's design. It was decided that the advantage of requiring such evaluations (in terms of increased program clarity) is worth the implementation burden. In practice, this means that implementers must evaluate such expressions using a rational arithmetic package (see "Universal Arithmetic Packages" by G. Fisher in ACM Ada Letters, Vol. 3, No. 6, pp. 30-47, May-June, 1984.) Consequently, it would not be consistent with the design intent and AI-00325 for an implementation to return FALSE as the value of 1.0/3.0 * 3.0 = 1.0. In general, the only acceptable reason for refusing to evaluate a static universal real expression exactly is insufficient memory to hold the required values.

Ada Commentary ai-00217-co.wj downloaded on Thu Aug 11 16:50:32 EDT 1988

| !standard 03.05.07 (09) 88-05-23 AI-00217/05
!class confirmation 86-11-14
| !status approved by WG9/AJPO 88-02-05
| !status approved by Director, AJPO 88-02-05
| !status approved by WG9/Ada Board 87-12-07
!status approved by Ada Board 87-07-30
!status panel/committee-approved 87-03-16 (reviewed)
!status panel/committee-approved (6-0-0) 86-11-14 (pending editorial review)
!status work-item 86-08-08
!status received 84-03-13
!references AI-00407, AI-00508, 83-00326
!topic The safe numbers of a floating point subtype

!summary 86-12-28

The safe numbers of a floating point subtype are the safe numbers of its base type.

!question 86-12-28

3.5.7(9) says:

> The safe numbers of a [floating point] subtype are those of its base type.

Shouldn't this say that the safe numbers of a floating point subtype are a SUBSET of those of its base type?

!response 87-03-13

The statement in 3.5.7(9) is correct because operations on values of a subtype are defined in terms of operations on safe numbers of the base type. Therefore, the safe numbers of a subtype should be (and are) the same as the base type's safe numbers.

Safe numbers are used (in the Standard) to specify the accuracy of real numeric operations. 4.5.7(8) defines the accuracy of all real numeric operations in terms of the safe numbers (since the model numbers of a type are a subset of the safe numbers and the rules for computing with safe numbers are the same as the rules for computing with model numbers). Since all numeric operations are declared for types rather than subtypes, these operations are performed using the safe numbers of the base type, and hence, in order to define the accuracy of operations on values of a subtype, it was sufficient (and intended) for the safe numbers of a floating point subtype to be the same as the safe numbers of its base type.

Ada Commentary ai-00231-ra.wj downloaded on Thu Aug 11 16:50:30 EDT 1988

```
!standard 03.08.01 (04)                                88-05-23  AI-00231/05
!class ramification 84-03-13
!status approved by WG9/AJPO 88-02-05
!status approved by Director, AJPO 88-02-05
!status approved by WG9/Ada Board 87-12-07
!status approved by Ada Board 87-07-29
!status panel/committee-approved 87-05-06 (reviewed)
!status panel/committee-approved (8-0-1) 87-02-17 (pending editorial review)
!status work-item 87-01-18
!status received 84-03-13
!references 83-00342
!topic Full declarations of incomplete types can have discriminants
```

!summary 87-01-18

The full declaration of an incomplete type can be a derived type with unconstrained discriminants when no discriminant part is given in the incomplete type's declaration.

!question 87-01-18

3.8.1(4) says:

> A discriminant part must be given in the full type declaration [for an incomplete type] if and only if one is given in the incomplete type declaration;

No further restriction is placed on the nature of the full type declaration, in contrast to the wording for private types (7.4.1(3)):

> If the private type declaration includes a discriminant part, the full declaration must include a discriminant part that conforms (see 6.3.1 for the conformance rules) and its type definition must be a record type definition. Conversely, if the private type declaration does not include a discriminant part, the type declared by the full type declaration (the FULL TYPE) must not be an unconstrained type with discriminants. The type must not be an unconstrained array type.

In particular, it appears that an incomplete type without a discriminant part can have a full declaration that is an unconstrained array type or a derived type with unconstrained discriminants:

```
        type REC (D : INTEGER) is
            record
                null;
            end record;

        type T;

        type T is new REC;       -- legal? (yes)
```

Was this intended?

!response 87-07-07

As noted in the question, the wording in 3.8.1(4) allows the full declaration of an incomplete type to be an unconstrained array type or a derived type that has unconstrained discriminants. This causes no difficulty, because prior to the end of the incomplete type's full type declaration, it can only be used as the type mark in the subtype indication of an access type definition (3.8.1(4)). Such usage is allowed for any unconstrained array type or type with discriminants, whether or not the discriminants have defaults; allowing it for incomplete types causes no problem.

Such full declarations would cause problems for private types, however, since it would be impossible to provide an appropriate constraint outside the package declaring the type.

Ada Commentary ai-00235-ra.wj downloaded on Thu Aug 11 16:50:20 EDT 1988

```
!standard 04.05.05 (11)                                 88-05-23  AI-00235/05
!class ramification 84-04-13
!status approved by WG9/AJPO 88-02-05
!status approved by Director, AJPO 88-02-05
!status approved by WG9/Ada Board 87-12-07
!status approved by Ada Board 87-07-30
!status panel/committee-approved 87-05-06 (reviewed)
!status panel/committee-approved (6-0-2) 87-02-18 (pending editorial review)
!status work-item 87-01-15
!status received 84-04-13
!references 83-00344
!topic Redundant parentheses enclosing universal_fixed expressions
```

!summary 87-01-15

An expression having type universal_fixed can be enclosed in parentheses before being converted to some other numeric type.

!question 87-05-05

4.5.5(11) requires that the result of either fixed point multiplication or fixed point division "must always be explicitly converted to some numeric type." It is not clear whether this means that such fixed point operations must occur syntactically as the immediate operand of a (numeric) type conversion, as in

```
DUR : DURATION;
...
DUR := DURATION(DUR*DUR);
```

or whether there are ANY other allowed syntactic variations. In particular, what about

```
DUR := DURATION((DUR*DUR));       -- are extra parens legal? (yes)
DUR := DURATION(((DUR/DUR)));     -- are extra parens legal? (yes)
```

!response 87-10-15

4.5.5(11) says:

> Multiplication of operands of the same or of different fixed point types is exact and delivers a result of the anonymous predefined fixed point type universal_fixed whose delta is arbitrarily small. The result of any such multiplication must always be explicitly converted to some numeric type. This ensures explicit control of the accuracy of the computation. The same considerations apply to division of a fixed point value by another fixed point value. No other operators are defined for the type universal_fixed.

Enclosing a multiplication or division in parentheses and then converting the parenthesized expression does satisfy the requirement to convert the result of these fixed point operations.

Ada Commentary ai-00245-ra.wj downloaded on Tue Aug 9 14:31:15 EDT 1988

!standard 06.04.01 (03) 88-05-23 AI-00245/08
!standard 08.05 (08)
!class ramification 84-05-14
!status approved by WG9/AJPO 88-02-05
!status approved by Director, AJPO 88-02-05
!status approved by WG9/Ada Board 87-12-07
!status approved by Ada Board 87-07-30
!status panel/committee-approved 87-05-06 (reviewed)
!status panel/committee-approved (9-0-0) 87-02-17 (pending editorial review)
!status work-item 85-02-04
!status received 84-05-14
!references AI-00318, 83-00363
!topic Type conversion conformance for renamed subprogram/entry calls

!summary 87-08-20

When a type conversion is used as an actual parameter corresponding to an IN OUT or OUT formal parameter and the subprogram being called was declared by a renaming declaration (renaming either a subprogram or entry), the name given as the type mark (in the type conversion) must conform to the name given for the corresponding parameter of the denoted subprogram or entry (not the name given in the renaming declaration).

!question 87-01-16

Section 8.5(8) states that the parameter subtypes in a subprogram renaming are those of the original subprogram declaration. Section 6.4.1(3) states that when a type conversion is used as an actual parameter corresponding to an IN OUT or OUT parameter, "the type mark must conform (see 6.3.1) to the type mark of the formal parameter." Which type mark is used in the case of a renamed subprogram? Consider:

```
OBJ : POSITIVE;
procedure PROC (X : in out INTEGER)...
procedure RENA (Y : in out POSITIVE) renames PROC;
...
RENA(INTEGER(OBJ));                   -- (1) Legal? (yes)
RENA(POSITIVE(OBJ));                  -- (2) Legal? (no)
```

Which of these calls is legal?

!response 87-01-18

8.5(8) says:

> The subtypes of the parameters and result (if any) of a renamed subprogram or entry are not affected by renaming. These subtypes are those given in the original subprogram dec- laration, generic instantiation, or entry declaration (not those of the renaming declaration); even for calls that use the new name.

This says the name for RENA's formal parameter subtype is INTEGER, the name given in the declaration of the denoted subprogram. For two simple names to conform, the visibility rules must give them the same meaning (6.3.1(5)), i.e., the names must refer to the same declaration (8.3(2-3)). Since INTEGER and POSITIVE are declared by different declarations, (2) is illegal.

Ada Commentary ai-00258-bi.wj downloaded on Tue Aug 9 14:24:30 EDT 1988

```
| !standard 13.07.02 (07)                                88-06-13  AI-00258/06
  !standard A        (34)
  !class binding interpretation 84-05-26
| !status approved by WG9/AJPO 88-02-05 (corrected in accordance with AI-00503)
| !status approved by WG9/AJPO 86-07-22
  !status approved by Director, AJPO 86-07-22
  !status approved by WG9/Ada Board 86-07-22
  !status approved by Ada Board 86-07-22
  !status WG14/ADA Board approved 84-11-27
  !status WG14-approved 84-11-27
  !status board-approved 84-11-26
  !status committee-approved 84-06-28
  !status work-item 84-06-12
  !status received 84-05-26
| !references 83-00368, 83-00860
  !topic 'POSITION etc. for renamed components
```

!summary 84-09-10

The prefix for 'POSITION, 'FIRST_BIT, and 'LAST_BIT must have the form R.C, where R is a name denoting a record and C is the name of a component of the record.

!question 84-09-10

Can a name declared by a renaming declaration be used with the 'POSITION, 'FIRST_BIT, and 'LAST_BIT attributes? The definitions in 13.7.2(7-10) all use the notation R.C, suggesting that the prefix of these attributes must have the form of a selected component whose prefix denotes a record, but Annex A(34) says that P'POSITION is allowed "for a prefix P that denotes a component of a record object". The wording in the annex allows a name declared by a renaming declaration as a prefix for 'POSITION, while the wording in 13.7.2 seems to disallow such usage. Which wording is correct?

!recommendation 84-09-10

The wording in the Annex is a summary of the actual definition, which is given in 13.7.2(7).

| !discussion 88-06-13

The preferred interpretation from an implementer's viewpoint is to require that the prefix have the form of a selected component whose prefix denotes a record, because of examples like the following:

```
        type R is record
             C1 : String (1..M);       -- M not compile-time determinable
             C2 : String (1..N);       -- N not compile-time determinable
        end record;

        type Acc_R is access R;
        function F return Acc_R is
        begin ... end F;

        package P is
             subtype STR_N is STRING(1..N);
             Obj      : R;
             Ren_C2   : STR_N renames F.all.C2;
        end P;
```

Now consider the following attributes:

```
        P.Obj.C1'Position    -- Easy to compute.
        P.Obj.C2'Position    -- Must be computed at run time since C1's
                             -- length is not static.
        F.all.C2'Position    -- Same as for P.Obj.C2'Position.
        P.Ren_C2'Position    -- Illegal?
```

The last case presents a problem because P.Ren_C2 would normally be implemented as a pointer to the second component of the object containing component C2. But the value of the P.Ren_C2'Position must be calculated using the address of the object containing C2. The code needed to calculate this address cannot usually be generated at the point where P.Ren_C2 is written. For example, one cannot reevaluate F.all in order to determine the object containing Ren_C2. So, because of the possibility of writing P.Ren_C2'Position, the elaboration of the renaming declaration must determine and save the address of F.all (i.e., of the object containing C2). This overhead would be incurred for every renaming of a record component.

It was not the intent to impose such an implementation overhead on renamings of record components. The wording in 13.7.2(7-10) justifies limiting the use of the 'Position, 'First_Bit, and 'Last_Bit attributes to those contexts in which the prefix has the form of a selected component whose prefix denotes a record. The wording in the Annex is not definitive.

Ada Commentary ai-00263-bi.wj downloaded on Thu Aug 11 16:50:19 EDT 1988

```
| !standard 03.02    (08)                                    88-05-23  AI-00263/06
  !standard 13.05    (04)
  !standard 13.07.03 (03)
  !standard 13.07.03 (05)
  !class binding interpretation 84-07-29
| !status approved by WG9/AJPO 88-02-05
| !status approved by Director, AJPO 88-02-05
| !status approved by WG9/Ada Board 87-12-07
  !status approved by Ada Board 87-07-30
  !status panel/committee-approved 87-05-06 (reviewed)
  !status panel/committee-approved (8-0-1) 87-02-17 (pending editorial review)
  !status work-item 87-01-14
  !status received 84-07-29
  !references 83-00391, 83-00843
  !topic A named number is not an object
```

!summary 87-03-22

A number declaration declares a named number, which is not an object.

The elaboration of a number declaration proceeds by evaluating the initialization expression and creating the named number. The value of the initialization expression then becomes the value of the named number.

!question 87-05-05

The Standard implies that named numbers are not objects, by explicitly mentioning named numbers in addition to objects when both are allowed (see, for example, 4.4(3)). However, 3.2(8) confuses the matter by stating that:

> A number declaration is a special form of object declaration ...

Is a named number an object? The distinction matters in a few places. For example, is an address clause allowed for a named number? Do the elaboration rules for object declarations apply to number declarations?

!recommendation 87-03-22

A number declaration declares a named number, which is not an object.

The elaboration of a number declaration proceeds by evaluating the initialization expression and creating the named number. The value of the initialization expression then becomes the value of the name number.

!discussion 87-07-07

3.2(1-2) states:

> An object is ... an object declared by an object declaration ...

Since an object declaration and a number declaration are distinct syntactic categories (see 3.2(9)), a named number is not an object even though 3.2(8) states:

> A number declaration is a special form of object declaration ...

and 3.2.2(1) states:

> A number declaration is a special form of constant declaration.

Since a named number is not an object, a named number cannot be given as the prefix for the attributes 'ADDRESS and 'SIZE, nor can an address clause be given for a named number.

Since a number declaration is not an object declaration, the elaboration of number declarations is not covered by the rules for elaborating object declarations. However, the elaboration rules for number declarations are clearly intended to be similar to those for object declarations.

| !standard 06.04.01 (04) 88-05-23 AI-00295/0
!class binding interpretation 84-10-16
| !status approved by WG9/AJPO 88-02-05
| !status approved by Director, AJPO 88-02-05
| !status approved by WG9/Ada Board 87-12-07
!status approved by Ada Board 87-07-30
!status panel/committee-approved 87-05-06 (reviewed)
!status panel/committee-approved (8-0-1) 87-02-17 (pending editorial review)
!status work-item 87-01-12
!status received 84-10-16
!references AI-00024, 83-00442
!topic Evaluating the variable in an actual parameter type conversion

!summary 87-03-13

For an actual parameter (of any type) of mode in out or out that is a typ
conversion, the variable name is evaluated before the call and therefor
determines the denoted entity.

!question 87-05-05

The first two sentences of 6.4.1(4) read:

> The variable name given for an actual parameter of mode in out or out is evaluated before the call. If the actual parameter has the form of a type conversion, then before the call, for a parameter of mode in out, the variable is converted to the specified type; after (normal) completion of the subprogram body, ...

The first sentence appears to apply only to actual parameters that are simpl
variable names, but not to actual parameters that are type conversions o
variable names. The second sentence mentions mode in out, but not mode out
implying by omission that type conversions for mode out are not performe
before the call. However, AI-00024 states that type conversions for mode ou
are performed before the call for array and access types (implying that th
variable name is evaluated); but its discussion section says that for recor
and scalar types, conversion before the call is not necessary.

Consider the following example:

```
declare
     type INT is new INTEGER;
     type ARR is array (1 .. 10) of INT;

     I : INTEGER := 2;
     A : ARR     := (others => 0);

     procedure P (J : out INTEGER; K : in out INTEGER) is
     begin
          K := K + 1;
          J := 10;
     end P;
```

```
        begin
             P (INTEGER (A (I)), I);
        end;
```

Unless the scalar variable A(I) is evaluated before the call, it is not clear whether A(2) or A(3) is updated.

For an actual parameter (of any type) of mode in out or out that is a type conversion, is the variable name evaluated before the call?

!recommendation 87-03-16

For an actual parameter of mode in out or out that is a type conversion, the variable name is evaluated before the call.

!discussion 87-08-20

It was intended that the first sentence of 6.4.1(4) include the case of variable names within type conversions; such variable names are to be evaluated before the call. So A(2) in the example is updated.

Ada Commentary ai-00305-bi.wj downloaded on Thu Aug 11 09:47:33 EDT 1988

| !standard 13.07.02 (03) 88-05-23 AI-00305/05
!class binding interpretation 84-10-16
| !status approved by WG9/AJPO 88-02-05
| !status approved by Director, AJPO 88-02-05
| !status approved by WG9/Ada Board 87-12-07
| !status approved by Ada Board 87-07-30
!status panel/committee-approved 87-05-06 (reviewed)
!status panel/committee-approved (7-1-1) 87-02-17 (pending editorial review)
!status work-item 87-01-14
!status received 84-10-16
!references 83-00439
!topic T'ADDRESS when T is a task type yields the task object address

!summary 87-03-22

| If T denotes a task type, then within the body of task unit T, the T in T'
| ADDRESS is considered to refer to the name of the task object that designates the task currently executing the body, i.e., T'ADDRESS returns the address of the object.

!question 87-03-22

Inside the body of a task type T, 9.1(4) says:

> the name of the corresponding task unit can also be used to refer to the task object that designates the task currently executing the body.

The 'ADDRESS attribute is defined both for objects and program units. Within the body of task type T, T can be used both to denote a task object and a task unit:

```
task type T;

task body T is
     X : INTEGER;
begin
     ... T.X := 5;          -- Use of T as unit name
     ... T'CALLABLE         -- Use of T as object name
     ... T'ADDRESS ...      -- Address of object or code?  (object)
end T;
```

If the T in T'ADDRESS is considered to be the name of the task unit, then the value returned should be the "machine code associated with the corresponding body" (13.7.2(3)). If it is considered to be the name of a task object, then it should return "the address of the first of the storage units allocated" to the object (13.7.2(3)). Which value does T'ADDRESS yield?

!recommendation 87-03-22

| If T denotes a task type, then within the body of task unit T, the T in T'
| ADDRESS is considered to refer to the name of the task object that designates the task currently executing the body.

T'ADDRESS when T is a task type yields the task object address AI-00305/05 2
88-05-23 BI WJ

!discussion 87-03-22

The current wording does not provide a clear basis for deciding which value should be returned. The intent, however, is that within the body of T, T refers to the task object that designates the task currently executing the body; therefore, within the body, T'ADDRESS should return the address of this task object. Outside the body, T can only refer to the task unit, never a task object, so T'ADDRESS should then return the address of T's machine code.

Ada Commentary ai-00306-bi.wj downloaded on Thu Aug 11 09:47:13 EDT 1988

!standard 13.09 (03) 88-05-23 AI-00306/15
!standard 02.08 (09)
!class binding interpretation 84-10-16
!status approved by WG9/AJPO 88-02-05
!status approved by Director, AJPO 88-02-05
!status approved by WG9/Ada Board 87-12-07
!status approved by Ada Board 87-07-30
!status panel/committee-approved 87-05-06 (reviewed)
!status panel/committee-approved (10-0-3) 87-02-17 (by ballot) (pending editorial review)
!status panel/committee-approved (7-0-2) 86-11-13 (pending letter ballot)
!status work-item 86-09-10
!status failed letter ballot (2-9-2) 86-09
!status committee-approved (8-0-1) 86-05-13 (pending letter ballot)
!status committee-approved (6-1-0) 86-02-21 (pending editorial review)
!status work-item 86-01-24
!status received 84-10-16
!references 83-00445, 83-00492, 83-00637, 83-00697, 83-00718, 83-00726, 83-00790
!topic Pragma INTERFACE: allowed names and illegalities

!summary 87-08-20

If a pragma INTERFACE names a language that is acceptable to an implementation, the subprogram name must denote one or more subprograms declared explicitly earlier in the same declarative part or package specification. (The pragma has no effect if no named subprogram satisfies the requirements.) The pragma is applied to all such subprograms other than enumeration literals and subprograms declared by generic instantiation.

If a subprogram named in the pragma was declared by a renaming declaration, the pragma applies to the denoted subprogram, but only if the denoted subprogram otherwise satisfies the above requirements.

It is illegal to apply a pragma INTERFACE to a subprogram for which a pragma INTERFACE has already been applied.

If a pragma INTERFACE applies to a subprogram, it is illegal to provide a body for the subprogram.

!question 86-07-01

The pragma INTERFACE is applied to a subprogram. Can the pragma be applied to any of the following subprograms?

a. enumeration literals

b. attributes that denote functions

c. predefined operators

d. derived subprograms

e. subprogram names declared by renaming declarations

f. subprograms declared by generic instantiations

g. subprograms declared by subprogram_body declarations

If an overloaded name given in a pragma INTERFACE denotes several subprograms of which only a few satisfy the requirements for the pragma, to which subprograms does the pragma apply, if any?

Is a pragma INTERFACE ignored if it names a subprogram that has a body, or is it illegal to provide such a body?

What is the effect if a pragma INTERFACE is given more than once for the same subprogram?

!recommendation 87-08-20

If a pragma INTERFACE names a language that is acceptable to an implementation, the subprogram name must denote one or more subprograms declared explicitly earlier in the same declarative part or package specification. The pragma is applied to all such subprograms other than enumeration literals and subprograms declared by generic instantiation.

It is illegal to apply a pragma INTERFACE to a subprogram for which a pragma INTERFACE has already been applied.

If a pragma INTERFACE applies to a subprogram, it is illegal to provide a body for the subprogram.

!discussion 86-08-20

The pragma INTERFACE gives a means of providing a subprogram body other than by a subprogram body declaration. The pragma was only intended to be applied to subprograms for which users can provide bodies. In particular, since an explicit subprogram body declaration cannot be provided for a subprogram that is an enumeration literal, an attribute, a predefined operator, or a derived subprogram, it was not intended that the pragma apply to such subprograms. For example:

```
declare
     type INT is range 1..10;
     pragma INTERFACE (FORTRAN, "+");    -- ignored
```

The pragma is ignored since the only subprogram "+" declared earlier in this declarative part is the implicitly declared predefined "+".

13.9(3) says that the subprogram name in the pragma INTERFACE is allowed to "stand for" several overloaded subprograms. Suppose the subprogram name stands for several subprograms, not all of which are declared earlier in the same declarative part or package specification:

```
        procedure P (B : BOOLEAN);        -- P.1
        package R is
             procedure P (I : INTEGER);   -- P.2
             pragma INTERFACE (XXX, P);
```

The intent is that the pragma INTERFACE be applied only to P.2 since P.2 is the only subprogram that is declared earlier in the same package specification; a body must be provided for P.1.

In addition, if a subprogram is overloaded, the pragma INTERFACE applies only to those subprograms for which bodies can be provided, e.g.:

```
        package P1 is
             type ENUM is (A, B, C);
             function B return INTEGER;
             pragma INTERFACE (XXX, B);
        end P1;
```

The pragma only applies to function B if it is supported for language XXX; it does not apply to enumeration literal B. Similarly:

```
        package T is
            function P return INTEGER;
        end T;

        package body T is
            procedure P;
            function P return INTEGER is
            begin ... end P;
            pragma INTERFACE (XXX, P);
        begin
            ...
```

In this case, the pragma only applies to procedure P since function P is not declared earlier in the same declarative part. (Function P is declared in the package specification.)

13.9(3) also says:

> A body is not allowed for such a subprogram (not even in the form of a body stub) since the instructions of the subprogram are written in another language.

This restriction means that if the pragma is accepted and is applied to certain subprograms, it is illegal to provide a body for any of these subprograms. For example:

```
        package P2 is
             procedure R (B : BOOLEAN);          -- R.1
             procedure R (I : INTEGER);          -- R.2
             pragma INTERFACE (YYY, R);          -- (1)
        end P2;
```

```
package body P2 is
     procedure R (B : BOOLEAN) is ...;  -- (2); illegal
end P2;
```

Since the pragma at (1) specifies that a body for R.1 and R.2 is supplied in language YYY, it is illegal to supply a body for either R.1 (as in (2)) or R.2. Similar illegalities can arise in a declarative part:

```
declare
     procedure R (B : BOOLEAN);         -- R.1
     procedure R (I : INTEGER);         -- R.2
     pragma INTERFACE (YYY, R);

     procedure R (B : BOOLEAN) is ...;  -- illegal
```

It is immaterial whether the pragma appears before or after the body:

```
declare
     procedure R (B : BOOLEAN);         -- R.1
     procedure R (I : INTEGER);         -- R.2
     procedure R (B : BOOLEAN) is ...;  -- illegal
     pragma INTERFACE (YYY, R);
begin
```

If the pragma is accepted for language YYY, the pragma applies to R.1 and R.2 so it is illegal to provide a body for R.1 (or R.2). It is similarly illegal to provide a body for R.1 even if the original declaration of R.1's specification is deleted:

```
declare
     procedure R (I : INTEGER);         -- R.2
     procedure R (B : BOOLEAN) is ...;  -- illegal body for R.1
     pragma INTERFACE (YYY, R);
begin
```

The pragma applies to subprogram R.1 as well as to R.2, since R.1 is declared explicitly (by the subprogram_body declaration). Since the subprogram body declaration also provides a body for R.1, and since it is intended to be illegal to provide a body for a subprogram to which the pragma applies, the declarative part is illegal.

Suppose the pragma INTERFACE is given more than once for the same subprogram:

```
package P2 is
     procedure P;
     pragma INTERFACE (L1, P);
     function P return INTEGER;
     pragma INTERFACE (L1, P);     -- illegal
end P2;
```

If L1 is an acceptable language, the second pragma would apply to procedure P as well as to function P. Since the pragma in effect provides a body for the subprograms to which it applies, and since two bodies cannot be given for the same subprogram, the second pragma is illegal.

If the subprogram named in the pragma was declared by a renaming declaration, the pragma applies to the denoted subprogram, but only if the denoted subprogram otherwise satisfies the requirements, i.e., the denoted subprogram must be explicitly declared earlier in the same declarative part or package specification and must not be an enumeration literal or a generic instance. The ability to use names declared by renaming declarations makes it easier to supply an Ada body for only one of two overloaded subprograms:

```
package P3 is
     function OVERLOADED return INTEGER;
     procedure OVERLOADED;
     procedure NO_BODY renames OVERLOADED;
     pragma INTERFACE (LANGUAGE, NO_BODY);
     ...
end P3;
```

The pragma applies just to the subprogram denoted by NO_BODY, i.e., the procedure. A body must still be provided for the function OVERLOADED.

Ada Commentary ai-00324-bi.wj downloaded on Thu Aug 11 09:47:12 EDT 1988

!standard 03.08 (06) 88-05-23 AI-00324/
!class binding interpretation 86-11-14
!status approved by WG9/AJPO 88-02-05
!status approved by Director, AJPO 88-02-05
!status approved by WG9/Ada Board 87-12-07
!status approved by Ada Board 87-07-30
!status panel/committee-approved 87-03-16 (reviewed)
!status panel/committee-approved (6-0-0) 86-11-14 (pending editorial review
!status panel/committee-approved (8-1-0) 86-09-10 (pending editorial review
!status work-item 86-01-23
!status received 85-01-26
!references 83-00494, 83-00806, 83-00826
!topic Checking the subtype of a non-null access value

!summary 86-01-28

An access value of type T belongs to every subtype of T if T's designat
type is neither an array type nor a type with discriminants.

!question 86-08-13

3.8(6) says:

> An access value belongs to a corresponding subtype of an access type either if the access value is the null value or if the value of the designated object satisfies the constraint.

3.2.1(15) says:

> The initialization of an object (...) checks that the initial value belongs to the subtype of the object; ....

5.2(3) says (for an assignment statement):

> A check is then made that the value of the expression belongs to the subtype of the variable, ...

The definition of compatibility of an access value and its subtype, and t
definitions of compatibility in object initializations and in assignme
statements apparently require an evaluation of the object designated by t
access value, and this suggests that all programs like the following a
erroneous:

```
declare
     type T is access INTEGER range 110..120;
     V : T := new INTEGER;  -- erroneous? (no)
begin
     V := new INTEGER;      -- erroneous? (no)
end;
```

In each case, the allocated object is not initialized and thus has
undefined value. Hence, V is assigned an access value whose designat
object's value may violate the range constraint 110..120. If so, shou
CONSTRAINT_ERROR be raised? Are these examples erroneous?

!recommendation 86-12-28

Every access value belongs to its corresponding access type. An access va
belongs to a subtype of an access type if the access value is the null va
or if the (access) subtype imposes no constraint on the designated object;
the (access) subtype imposes a constraint (such a constraint is possible c
if the designated type is an array type or a type that has a discriminar
the access value belongs to the access subtype if the value of the designa
object satisfies the constraint.

!discussion 86-10-13

3.8(6) defines what it means for an access value to "belong" to a subtype
an access type in the case where a constraint has been imposed on the acc
type (since 3.8(6) mentions "the constraint"). Such a constraint car
imposed only when the designated type is an array type or a type w
discriminants. 3.8(6) does not define what it means for an access value
"belong" to an access type or subtype when no constraint is imposed, as
the case when the designated type is a scalar type (since no range constra
can then be imposed on the access type). The intent, however, is clear: w
the access type is unconstrained, every access value of the type belongs
the access type, and to any of its subtypes. In particular, this holds w
the designated type is a scalar type or a private type without discriminar
In such cases, there is no need to check that the value of the designa
object, if any, satisfies a constraint. Consequently, the examples giver
the question are not erroneous.

Ada Commentary ai-00336-ra.wj downloaded on Tue Aug 9 16:58:21 EDT 1988

```
| !standard 13.05    (05)                          88-05-23  AI-00336/C
  !class ramification 86-04-11
| !status approved by WG9/AJPO 88-02-05
| !status approved by Director, AJPO 88-02-05
| !status approved by WG9/Ada Board 87-12-07
| !status approved by Ada Board 87-07-30
  !status panel/committee-approved 87-05-06 (reviewed)
  !status panel/committee-approved (7-0-2) 87-02-17 (pending editorial review)
  !status work-item 86-04-11
  !status received 85-05-02
  !references 83-00535
  !topic Address clauses for subprogram bodies
```

!summary 86-04-16

An address clause cannot be given for a subprogram whose body acts as it
declaration.

| !question 87-12-07

Can an address clause be given for a subprogram whose body serves as it
declaration? For example:

```
-- Example 1
procedure P;
for P use at ...;        -- legal
...
procedure P is ... end P;

-- Example 2
procedure Q is ... end Q;
for Q use at ...;        -- legal? (no)
```

!response 87-03-16

A representation clause is a basic declarative item but not a late
declarative item (3.9(2)). This means a representation clause cannot appe
after a body in a declarative part; to give an address clause for
subprogram, the clause must precede the subprogram's body. (This means ther
must be an explicit declaration that precedes the body, even if such
declaration is not otherwise needed.)

```
!standard 13.10.01 (06)                                  88-05-23  AI-00356/08
!standard 04.08    (07)
!standard 03.02.01 (18)
!class binding interpretation 85-06-18
!status approved by WG9/AJPO 88-02-05
!status approved by Director, AJPO 88-02-05
!status approved by WG9/Ada Board 87-12-07
!status approved by Ada Board 87-07-30
!status panel/committee-approved 87-03-16 (reviewed)
!status panel/committee-approved (8-0-0) 86-11-13 (pending editorial review)
!status work-item 86-05-20
!status returned to committee by WG9 86-05-09
!status committee-approved (9-0-0) 85-11-20
!status received 85-06-18
!references 83-00555
!topic Access values that designate deallocated objects
```

!summary 87-03-13

The storage occupied by a designated object can be reclaimed immediately after applying an instance of the unchecked deallocation procedure to an access variable that designates the object.

If two objects having non-null access values designate the same object and an instance of the unchecked deallocation procedure is applied to one of the objects, the other object is considered to have an undefined value; any attempt to use such a value makes execution of the program erroneous.

Similarly, if a name declared by a renaming declaration denotes a subcomponent of an object that is later freed by calling an instance of the unchecked deallocation procedure, the name is considered to have an undefined value; any attempt to evaluate the name (e.g., by assigning a value to it) makes execution of the program erroneous.

!question 86-07-30

Consider the following code fragment:

```
        Y := X;
        FREE(X);
        X := new CELL;
        if X = Y then ... end if;
```

The value of Y is not altered by the call to FREE. Therefore, Y will designate the same object after the call as it did before the call. Moreover, since Y still designates the object, the space occupied by the object cannot be reclaimed immediately (4.8(7)):

> An implementation must guarantee that any object created by the evaluation of an allocator remains allocated for as long as this object or one of its subcomponents is accessible directly or indirectly, that is, as long as it can be denoted by some name.

Since the space occupied by Y.all cannot be reclaimed immediately, the allocator, new CELL, must return an access value that is not equal to Y, and the result of the comparison must be FALSE. (Note that the Standard nowhere states that comparing two access values involves accessing the objects they designate, so 13.10.1(6) cannot be invoked to make the program erroneous.)

13.10.1(5) indicates that calling the unchecked storage deallocation procedure allows the storage occupied by a designated object to be reclaimed. But because 4.8(7) forbids reclaiming storage while it can still be accessed, it appears that the unchecked deallocation procedure does not allow immediate reclamation of storage. Was this the intent?

!recommendation 86-12-28

The storage occupied by a designated object can be reclaimed immediately after applying an instance of the unchecked deallocation procedure to an access variable that designates the object.

If two objects having non-null access values designate the same object and an instance of the unchecked deallocation procedure is applied to one of the objects, the other object is considered to have an undefined value; any attempt to use such a value makes execution of the program erroneous.

Similarly, if a name declared by a renaming declaration denotes a subcomponent of an object that is later freed by calling an instance of the unchecked deallocation procedure, the name is considered to have an undefined value; any attempt to evaluate the name makes execution of the program erroneous

!discussion 86-12-28

13.10.1(5) says:

> FREE(X), when X is not equal to NULL, is an indication that the object designated by X is no longer required, and that the storage it occupies is to be reclaimed.

It was intended that the unchecked deallocation procedure allow the storage occupied by a designated object to be reclaimed immediately, even if the designated object, or one of its subcomponents, can be denoted by some name. (A subcomponent can be denoted by a name declared by a renaming declaration.)

Given the intent to allow immediate storage reclamation, it was also intended that the programmer be responsible for avoiding use of names that denote objects (or subcomponents of objects) whose storage has been marked for reclamation by the unchecked deallocation procedure. 13.10.1(6) expresses this intent in terms of actually attempting to access the deallocated object. It was an oversight that this rule does not cover all uses of such access values, e.g., in comparisons, and for names that denote subcomponents of a deallocated object. In particular, for a name declared by a renaming declaration and denoting a subcomponent of a deallocated object, an attempt to assign to the subcomponent should be considered erroneous even though no attempt is being made to use the VALUE of the subcomponent. Hence, the recommendation states that an attempt to EVALUATE the name is erroneous. On the other hand, it is not erroneous to assign a new access value to a

variable that currently denotes a deallocated object; only an attempt to US
the value of such a variable is erroneous.

Ada Commentary ai-00366-ra.wj downloaded on Wed Aug 10 17:03:39 EDT 1988

!standard 13.07.01 (07) 88-05-23 AI-00366/07
!class ramification 86-07-30
!status approved by WG9/AJPO 88-02-05
!status approved by Director, AJPO 88-02-05
!status approved by WG9/Ada Board 87-12-07
!status approved by Ada Board 87-07-30
!status panel/committee-approved (10-0-3) 87-02-17 (by ballot)
!status panel/committee-approved (5-0-0) 86-11-14 (pending letter ballot)
!status work-item 86-07-30
!status received 85-07-21
!references AI-00201, 83-00581
!topic The value of SYSTEM.TICK for different execution environments

!summary 86-12-15

SYSTEM.TICK should have a value that reflects the precision of the clock in the main program's execution environment. If SYSTEM.TICK does not have an appropriate value, the effect of executing the program is not defined.

!question 86-12-15

Suppose an implementation's execution environment provides a basic clock period whose accuracy is dependent on the electrical line frequency (1/60 of a second in the USA or 1/50 of a second elsewhere, respectively). What should the value of SYSTEM.TICK be? Must an implementation supply different SYSTEM packages depending on clock precision, even when the clock precision is determined by the execution environment's line frequency?

Since SYSTEM.TICK is a constant, what happens to an Ada program that is compiled in the USA (and gets a SYSTEM.TICK of 1.0/60.0) if the executable image is brought to Europe?

!response 86-12-15

The value for SYSTEM.TICK should reflect the precision of the clock in the expected execution environment (AI-00201). If a program image is executed in an environment in which SYSTEM.TICK does not have the correct value, the effect is not defined by the language, any more than the effect is defined if an attempt is made to execute a program in an execution environment that is in some other way incompatible with the assumptions made when the program was compiled.

Ada encourages the portability of Ada programs at the source level and not at the executable image level. If two execution environments are identical, an Ada program could be ported in its executable image form. Otherwise, the program should be compiled for (or cross-compiled to) the intended target execution environment.

Ada Commentary ai-00367-bi.wj downloaded on Tue Aug 9 14:31:16 EDT 1988

```
!standard 03.04    (11)                              88-05-23  AI-00367/06
!standard 12.01    (05)
!class binding interpretation 86-07-31
!status approved by WG9/AJPO 88-02-05
!status approved by Director, AJPO 88-02-05
!status approved by WG9/Ada Board 87-12-07
!status approved by Ada Board 87-07-30
!status panel/committee-approved 87-01-19 (reviewed)
!status panel/committee-approved (10-0-0) 86-11-13 (pending editorial review)
!status work-item 86-07-31
!status received 85-07-30
!references AI-00398, 83-00589
!topic Deriving from types declared in a generic package
```

!summary 86-10-10

The rules concerning derivable subprograms in the visible part of a nongeneric package are applicable in the visible part of a generic package. (The effect of a derived type declaration in an instance of a generic unit is discussed in AI-00398.)

!question 85-07-30

Consider the following example:

```
        generic
        package Z is
             type T is (ALPHA, BETA);
             procedure P (X : T);
        private
             type DT is new T;
        end Z;
```

Is subprogram P derived for type DT? The parent type, T, is declared in the visible part of a generic package, and the Standard makes a clear distinction between a package and a generic package, e.g., 7(1) and 12(1). The rules explaining which subprograms are derivable state that certain subprograms are derivable when a parent type is declared in the visible part of a package (3.4(11)). Since parent type T in the above example is declared in the visible part of a generic package, these rules seemingly do not apply. Is it the intent that the rules concerning derivable subprograms be considered to apply to the visible part of generic packages as well as the visible part of nongeneric packages?

!recommendation 85-07-30

The rules given in 3.4(11) apply when the parent type of a derived type definition is declared in the visible part of a generic package.

!discussion 87-12-07

It was intended that when a parent type in a derived type definition is declared in the visible part of a generic package, the rules given in 3.4(11) for types and subprograms declared in nongeneric packages also apply to (parent) types and subprograms declared in generic packages. (The visibility rules only allow such type derivations within the generic unit itself, since no type declared in the visible part of a generic package is visible outside the generic unit, either directly or by selection.)

!standard 03.02.01 (18) 88-05-23 AI-00374/
!class binding interpretation 85-08-22
!status approved by WG9/AJPO 88-02-05
!status approved by Director, AJPO 88-02-05
!status approved by WG9/Ada Board 87-12-07
!status approved by Ada Board 87-07-30
!status panel/committee-approved 87-01-19 (reviewed)
!status panel/committee-approved (8-0-0) 86-11-13 (pending editorial reviev
!status work-item 86-10-10
!status received 85-08-22
!references 83-00616, 83-00621
!topic An attempt to access an undefined constant is erroneous

!summary 86-12-03

The execution of a program is erroneous if it attempts to evaluate a sca
CONSTANT with an undefined value.

!question 87-01-19

3.2.1(18) states:

> The execution of a program is erroneous if it attempts to evaluate a scalar variable with an undefined value.

The following fragment evaluates a scalar constant with undefined value:

```
type REC is
    record
        UNDEFINED : INTEGER;
    end record;
R1 : REC;                        -- (1)
R2 : constant REC := R1;         -- (2)
...
... R2.UNDEFINED ...             -- (3)
```

R1.UNDEFINED is undefined at (1). Since R1 is non-scalar, evaluating R1
not erroneous at (2). R2.UNDEFINED is undefined and used in an expression
(3), but 3.2.1(18) does not apply, since R2.UNDEFINED is a constant, no
variable. Was this the intent?

!recommendation 86-12-03

The execution of a program is erroneous if it attempts to evaluate a sca
CONSTANT with an undefined value.

!discussion 86-10-13

Evaluation of a scalar object having an undefined value was intended to
erroneous. Since constants must be initialized, 3.2.1(18) was writ
assuming that all constants have defined values. The example shows this
not the case, so, to satisfy the intent, 3.2.1(18) should be understood
apply as well to an attempt to evaluate a scalar constant with an undefi
value.

Ada Commentary ai-00375-ra.wj downloaded on Wed Aug 10 17:01:47 EDT 1988

```
!standard 03.05.07 (17)                                88-05-23  AI-00375/05
!class ramification 85-08-22
!status approved by WG9/AJPO 88-02-05
!status approved by Director, AJPO 88-02-05
!status approved by WG9/Ada Board 87-12-07
!status approved by Ada Board 87-07-30
!status panel/committee-approved 87-03-12 (reviewed)
!status panel/committee-approved (6-0-0) 86-11-14 (pending editorial review)
!status work-item 86-10-05
!status received 85-08-22
!references AI-00407, AI-00174, 83-00611
!topic Restricting the allowed values of a floating point subtype
```

!summary 86-12-28

If a floating point constraint in a subtype indication includes a range constraint, the range of values that belong to the subtype (i.e., that satisfy the constraint) is defined by the range constraint. If no range constraint is present, the range of values that belong to the subtype is not affected, even though the accuracy of the subtype may be reduced.

!question 86-12-28

The Note 3.5.7(17) reads in part:

> The imposition of a floating point constraint on a type mark in a subtype indication cannot reduce the allowed range of values unless it includes a range constraint (the range of model numbers that correspond to the specified number of digits can be smaller than the range of numbers of the type mark).

What is meant by reducing "the allowed range of values"?

!response 87-03-13

Consider the following declarations:

```
type T is digits 5 range 12345.0 .. 56789.0;
subtype ST is T digits 3;
```

The purpose of the note is to point out that subtype ST has the same range constraint as type T, i.e., the floating accuracy definition does not change the set of values that belong to subtype ST. In particular, it is the case that T'LAST and ST'LAST have the same model interval (as do T'FIRST and ST'FIRST), since these attributes yield values having ST's type (3.5(7-9)). In this example, 12345.0 and 56789.0 are model numbers of type T, so the model intervals contain a single value. Neither ST'FIRST nor ST'LAST is a model number of subtype ST, since ST's model numbers are redefined by the floating accuracy definition (3.5.7(15)) and the exact representation of either value requires more mantissa bits than are allowed for ST's model numbers. However, this potential inaccuracy in the representation of 12345.0 or 56789.0 is only of importance when attempting to assign one of these values to a variable having subtype ST (see AI-00407).

Ada Commentary ai-00388-ra.wj downloaded on Tue Aug 9 14:24:28 EDT 1988

```
!standard 02.08    (04)                              88-06-13  AI-00388/06
!class ramification 85-10-29
!status approved by WG9/AJPO 88-02-05 (corrected in accordance with AI-00511)
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status committee-approved (7-0-2) 85-11-22
!status work-item 85-10-29
!status received 85-09-16
!references 83-00298, 83-00635, 83-00871
!topic Pragmas are allowed in a generic formal part
```

!summary 85-10-10

Pragmas are allowed in a generic formal part.

!question 85-10-10

2.8(4) says:

> [Pragmas are allowed] after a semicolon delimiter, but not within a formal part or discriminant part.

In addition, 2.8(5) says:

> [Pragmas are allowed] at any place where the syntax rules allow a construct defined by a syntactic category whose name ends with "declaration", ...

Since a generic_formal_part is different from a formal_part, and since a generic formal part consists of a sequence of generic_parameter_declarations, does this mean that pragmas ARE allowed in a generic formal part, or was it the intent to forbid pragmas in generic formal parts as well?

!response 88-06-13

As noted in the question, the Standard does allow pragmas to appear in generic formal parts. In particular,

```
generic
    pragma PAGE;
    X : INTEGER;
    pragma PAGE;
procedure P;
```

is legal.

Pragmas are terminated with semicolons and are only allowed in contexts where semicolons are used to terminate constructs. Although a generic formal part, a formal part, and a discriminant part have similar functions, they use the semicolon differently. In a generic formal part, the semicolon appears at the end of each declaration. In a formal part and discriminant part, the semicolon separates declarations. In particular, a semicolon does not follow the last declaration. If a pragma were allowed after the last declaration, it would not be separated from the preceding declaration by a semicolon, since no semicolon is present. A special rule would be needed either to insert a semicolon in such a case or to forbid pragmas after the last declaration. Rather than provide a special rule, it was decided to disallow pragmas in contexts where semicolons separate constructs, namely in formal parts and discriminant parts.

Ada Commentary ai-00407-bi.wj downloaded on Wed Aug 10 17:01:27 EDT 1988

```
!standard 04.05.07 (00)                                88-05-23  AI-00407/06
!standard 03.05.08 (16)
!standard 03.05.10 (15)
!standard 05.02    (03)
!class binding interpretation 85-12-29
!status approved by WG9/AJPO 88-02-05
!status approved by Director, AJPO 88-02-05
!status approved by WG9/Ada Board 87-12-07
!status approved by Ada Board 87-07-30
!status panel/committee-approved (10-0-3) 87-02-17 (by ballot)
!status panel/committee-approved (3-0-3) 86-11-14 (pending letter ballot)
!status work-item 86-10-05
!status received 85-12-29
!references 83-00691
!topic The operations of a subtype with reduced accuracy
```

!summary 87-01-19

When assigning a fixed or floating point value to a variable, the stored value need only be represented as a model number of the variable's subtype. Furthermore, if no exception is raised by the assignment, the stored value belongs to the subtype of the variable.

If a real subtype is used as the type mark in a membership test, qualification, or explicit conversion, the corresponding operation is performed with the accuracy of the base type and the range of the subtype.

For a real subtype, the value of the attribute FIRST or LAST is represented with at least the accuracy of the base type. The values of other attributes of a real subtype are given exactly.

!question 86-10-05

3.5.8(16) says:

> The operations of a subtype are the corresponding operations of the type except for the following: assignment, membership tests, qualification, explicit conversion, and the attributes of the first group [BASE, FIRST, LAST, SIZE, DIGITS, MANTISSA, EPSILON, EMAX, SMALL, and LARGE]; the effects of these operations are redefined in terms of the subtype.

3.5.7(15) says:

> The elaboration of [a subtype indication consisting of a type mark followed by a floating point constraint] ... creates a floating point subtype whose model numbers are defined by the corresponding floating accuracy definition.

What do these two paragraphs together imply about the accuracy with which the assignment, membership tests, qualification, explicit conversion, and attribute operations are performed? For example, consider:

```
type T is digits 5;
subtype ST is T digits 3 range 12345.0 .. 15099.0;

X : ST := 12345.0;
```

What does "redefining the effect" of assignment in terms of subtype ST mean here? The effect of assignment is defined in 5.2(3):

> For the execution of an assignment statement, the variable name and the expression are first evaluated ... A check is then made that the value of the expression belongs to the subtype of the variable ... Finally, the value of the expression becomes the new value of the variable.

In the case of X's initialization, evaluation of the expression means implicitly converting 12345.0 to T's base type and checking that the converted value belongs to T's range. Since T's base type has at least 5 digits of accuracy, 12345.0 is a model number. This value certainly belongs to T's subtype, so it is then assigned to X. Or was the intent to allow the value of X to be approximated as a model number for ST rather than requiring that the stored value have at least 5 digits of accuracy, i.e., should one understand "becomes the new value of the variable" to mean "is allowed to become a model number of the variable's subtype?" If so, this would seem to allow the approximation to a model number of ST to occur after the range check has been performed, and this is too late. 12345.0 belongs to the model interval 12344.0 .. 12352.0 (since ST'MANTISSA = 11, and 12345.0 is 16#3039.0#). If 12345.0 is approximated as 12344.0 and becomes the value of X, can the expression, X in ST, be FALSE after the assignment? Can 12344.0 in ST evaluate to TRUE? Can it be the case that ST'(12345.0) will raise CONSTRAINT_ERROR? If the initialization expression for X were ST'FIRST instead of the literal 12345.0, could CONSTRAINT_ERROR be raised? In short, what does it mean for the "effects" of the operations listed in 3.5.8(16) to be "redefined in terms of the subtype?"

Note that the same questions arise for fixed point types, although for fixed point, the relevant attributes are BASE, FIRST, LAST, SIZE, DELTA, MANTISSA, SMALL, LARGE, FORE, and AFT.

!recommendation 86-10-05

If the subtype of the variable in an assignment statement has less accuracy than the type of the expression, the model interval of the expression is widened to the smallest containing model interval of the subtype. Any value in the widened model interval can become the value of the variable, but CONSTRAINT_ERROR is raised if the value to be stored does not belong to the variable's subtype.

If a real subtype is used as the type mark in a membership test, qualification, or explicit conversion, the corresponding operation is performed with the accuracy of the base type and the range of the subtype.

For a real subtype, the value of the attribute FIRST or LAST is represented with at least the accuracy of the base type. The values of other attributes of a real subtype are given exactly.

!discussion 87-08-20

3.5.7(10-12) says that the declaration:

```
        type T is digits 5;
```

is equivalent to:

```
        type 'floating_point_type' is new predefined_floating_point_type;
        subtype T is 'floating_point_type' digits 5;
```

The only operations declared for type T are those declared for the base type -- no operations are declared by the subtype declaration itself. Consequently, all operations for type T are performed using the safe numbers of the base type. Suppose T'BASE'DIGITS = T'DIGITS, i.e., the base type has exactly the accuracy of the subtype. Now consider the effect of the declaration of subtype ST:

```
        subtype ST is T digits 3 range 12345.0 .. 15099.0;
```

No new operations are declared. Let's consider, however, the effect of evaluating a conversion to subtype ST:

```
        ST(12345.0)
```

According to 4.6(4), the universal_real value, 12345.0, is first converted to ST's base type. The converted value is then checked to see if it belongs to subtype ST. This check is performed using the predefined operations of the base type, i.e., the check is performed with the accuracy of ST's base type. The only sense in which the effect of converting to subtype ST is different from the effect of converting to the base type is that ST's range is used instead of the range associated with T's base type. Since 12345.0 is a model number of T's base type, the conversion is performed exactly. Since ST's range constraint is specified with model numbers of type T, the check to see if the converted value belongs to subtype ST is equivalent to evaluating:

```
        12345.0 in 12345.0 .. 15099.0
```

This evaluation must yield TRUE, so no exception can be raised by the conversion. The fact that ST has reduced accuracy does not affect how the conversion is performed. Similar reasoning applies for qualification.

Now consider the membership test, 12345.0 in ST. Since the membership test operation is declared for T's base type, it is performed with the accuracy of the base type, but uses the range constraint associated with subtype ST. As for conversion and qualification, the reduced accuracy of subtype ST does not affect how the membership test is evaluated.

In short, for membership tests, conversion, and qualification, redefining the effect of these operations in terms of the subtype means using the range of the subtype but otherwise using the operations of the base type.

A similar conclusion could be derived for the assignment operation, but this was not the intent. In some implementations, floating point values with more than, say, N digits of accuracy require a double precision representation. If a variable has a subtype with less than N digits of accuracy, the intention was that single precision would suffice to hold stored values of the variable. To achieve this intent and to ensure that the stored value still satisfies the variable's range constraint after conversion to a less precise representation, it is essential that the less precise value (the value that will actually be stored) be checked against the subtype's range before the assignment is actually done.

For floating point subtypes, the attributes affected by a subtype declaration are given in the question. The attributes DIGITS, MANTISSA, EPSILON, EMAX, SMALL, and LARGE return values that depend on the accuracy specified for the subtype, and in this sense, the effect of these attributes is defined by the subtype declaration. Since the values returned by EPSILON, SMALL, and LARGE are model numbers of the subtype, no inaccuracy is allowed in evaluating these attributes.

The attributes FIRST and LAST are declared as operations of the base type. When applied to a subtype, the returned value is therefore a value of the base type, although the value actually returned depends on the range specified for the subtype. Hence, for the example given in the question, since 12345.0 is a model number of the base type, ST'FIRST is exactly equal to 12345.0, even though ST'FIRST is not a model number of subtype ST.

Similar reasoning applies to fixed point operations.

!standard 04.01.03 (15) 88-05-23 AI-00412/06
!standard 04.01.03 (18)
!standard 12.01 (05)
!class binding interpretation 86-03-06
!status approved by WG9/AJPO 88-02-05
!status approved by Director, AJPO 88-02-05
!status approved by WG9/Ada Board 87-12-07
!status approved by Ada Board 87-07-30
!status panel/committee-approved 87-05-06 (reviewed)
!status panel/committee-approved (10-0-0) 86-11-13 (pending editorial review)
!status work-item 86-08-08
!status received 86-03-06
!references AI-00504, 83-00716
!topic Expanded names for generic formal parameters

!summary 87-03-13

A formal parameter of a generic unit can be denoted by an expanded name.

!question 87-03-13

12.1(5) says:

> Within the declarative region associated with a generic subprogram, the name of this program unit denotes the subprogram obtained by the current instantiation of the generic unit. Similarly, within the declarative region associated with a generic package, the name of this program unit denotes the package obtained by the current instantiation.

4.1.3(17) says:

> The prefix [of an expanded name] must denote a construct that is ... a program unit, ... The selector must be the simple name, character literal, or operator symbol of an entity whose declaration occurs immediately within the construct.

Now consider the following example:

```
generic
    FORMAL : INTEGER;
package P is
    Z : INTEGER := P.FORMAL;  -- legal? (yes)
end P;
```

The expanded name P.FORMAL is allowed by 4.1.3(17) if P is considered to denote the enclosing generic unit, but 12.1(5) says P denotes the package obtained by the current instantiation, and FORMAL is not declared within any instance of package P. Is P.FORMAL an allowed name?

!recommendation 87-03-13

Within a generic unit, a simple name or operator symbol declared in a generic formal part can be the selector of an expanded name whose prefix is either the simple name of the unit or an expanded name whose selector is the simple name of the unit. Within the generic unit, such an expanded name denotes the corresponding generic formal parameter.

!discussion 87-03-13

It was the intent to allow expanded names for all entities declared immediately within a program unit. Consequently, it was intended to allow a generic formal parameter to serve as the selector in an expanded name denoting the formal parameter, even though the prefix of such an expanded name is considered to denote "the current instantiation." (In an instance, such an expanded name denotes the entity corresponding to the formal parameter, as specified by 12.3(6-12).)

Ada Commentary ai-00422-bi.wj downloaded on Wed Aug 10 17:01:04 EDT 1988

```
!standard 13.01    (03)                             88-05-23  AI-00422/06
!standard 13.03    (02)
!standard 13.04    (02)
!class binding interpretation 86-05-22
!status approved by WG9/AJPO 88-02-05
!status approved by Director, AJPO 88-02-05
!status approved by WG9/Ada Board 87-12-07
!status approved by Ada Board 87-07-30
!status panel/committee-approved 87-03-16 (reviewed)
!status panel/committee-approved (6-0-0) 86-11-14 (pending editorial review)
!status work-item 86-10-13
!status received 86-05-22
!references 83-00742, 83-00842
!topic Representation clauses for derived enumeration and record types
```

!summary 87-01-28

An enumeration representation clause or a record representation clause can be given for an enumeration type or a record type declared by a derived type declaration.

The index subtype for the aggregate used in an enumeration representation clause is the base type of the enumeration type.

A record representation clause for a first named record subtype can specify the representation of any component that belongs to the record's base type, even if the subtype is constrained.

!question 87-01-28

3.3.1(4) says:

> ... the elaboration of the type definition for a numeric or derived type creates both a base type and a subtype of the base type.

3.3.1(5) further says:

> The simple name declared by a full type declaration denotes the declared type, unless the type declaration declares both a base type and a subtype of the base type, in which case the simple name denotes the subtype, and the base type is anonymous.

It is clear, therefore, that the name declared by a derived type declaration denotes a subtype, not a type. Now 13.1(3) says:

> a first named subtype is ... a subtype declared by a type declaration, the base type being therefore anonymous. ... An enumeration representation clause is only allowed for an enumeration type; a record representation clause, only for a record type.

Since this paragraph defines the difference between a "type" and a "first named subtype," it is especially significant that the rule for enumeration and record representation clauses only uses the word type. The clear conclusion is that an enumeration representation clause and a record representation clause cannot be given for a derived enumeration type or a derived record type. This does not seem to be the intent, however, since 13.6(2) gives an example of a record representation clause for a derived record type.

Assuming that the intent was to allow a representation clause for a derived record or enumeration type when there is no constraint imposed on the derived type, can a record representation clause or enumeration representation clause be given if the derived type declaration includes a constraint? Consider the following declarations:

```
type REC (D : POSITIVE) is
     record
          case D is
               when 1..10 =>
                    C1 : INTEGER range 0..15;
               when others =>
                    C2 : STRING (1..10);
          end case;
     end record;

type D_REC is new REC (3);
for D_REC use ...;
```

Should the record representation clause only mention the components that occur in the subtype?

For derived enumeration types, similar questions arise. For example, consider:

```
type ENUM is (A, B, C);
type D_ENUM is new ENUM range A..B;
for D_ENUM use ...;
```

It is unclear what array aggregate can be written for D_ENUM's representation clause. In the above case, can the aggregate only specify the representation for literals A and B?

!recommendation 87-01-28

A record representation clause can be given for a first named record subtype.

An enumeration representation clause can be given for a first named enumeration subtype.

!discussion 87-03-13

13.6(1) says:

> At most one representation clause is allowed for a given type and a given aspect of its representation. Hence, if an alternative representation is needed, it is necessary to declare a second

type, derived from the first, and to specify a different representation for the second type.

This wording, plus the example given in this section, show that the intent was to allow a record representation clause to be given for a derived record type. If the derived type is a subtype of the parent type (i.e., if a constraint was imposed by the derived type definition), the record representation clause can nonetheless be given for the full type (since 13.4(6) speaks of allowing "at most one component clause ... for each component of the record TYPE", i.e., for each component of the base type).

Similarly, it was the intent to allow an enumeration representation clause to be given for a derived enumeration type. Any constraint given in the derived type declaration is to be ignored when giving the enumeration clause for the derived type, since 13.3(3) says the aggregate's index subtype is the enumeration type, i.e., the base type of the derived type.

Ada Commentary ai-00430-bi.wj downloaded on Tue Aug 9 14:29:48 EDT 1988

```
| !standard 03.09    (08)                                88-05-23  AI-00430/05
  !standard 03.05.01 (03)
  !class binding interpretation 87-02-17
| !status approved by WG9/AJPO 88-02-05
| !status approved by Director, AJPO 88-02-05
| !status approved by WG9/Ada Board 87-12-07
| !status approved by Ada Board 87-07-30
  !status panel/committee-approved 87-05-06 (reviewed)
  !status panel/committee-approved (9-0-0) 87-02-17 (pending editorial review)
  !status work-item 87-01-19
  !status received 86-06-19
  !references 83-00748
| !topic Using an enumeration literal does not raise PROGRAM_ERROR
```

!summary 87-01-21

The use of an enumeration literal (i.e., a call of the corresponding parameterless function) does not raise PROGRAM_ERROR.

!question 87-01-19

3.5.1(3) states:

> Each enumeration literal specification is the declaration of the corresponding enumeration literal: this declaration is equivalent to the declaration of a parameterless function, the designator being the enumeration literal, and the result type being the enumeration type. The elaboration of an enumeration type definition ... includes that of every enumeration literal specification.

The elaboration of an enumeration type definition includes the elaboration of a function declaration for each enumeration literal; but there is no elaboration of the corresponding function bodies. Will the use of an enumeration literal (i.e., a function call) in an expression raise PROGRAM_ERROR (3.9(5, 8))?

!recommendation 87-01-21

The implicitly declared function body for an enumeration literal is elaborated when the corresponding enumeration literal specification is elaborated.

!discussion 87-01-21

The declaration of an enumeration literal is equivalent to the declaration of a parameterless function. The body of this function is declared implicitly and must be elaborated. To ensure calls of an enumeration literal will not raise PROGRAM_ERROR, it is sufficient and reasonable to assume that the body is elaborated when the corresponding enumeration literal specification is elaborated.

Ada Commentary ai-00431-ra.wj downloaded on Tue Aug 9 16:53:16 EDT 1988

!standard 04.05.01 (03) 88-05-23 AI-00431/05
!class ramification 87-01-21
!status approved by WG9/AJPO 88-02-05
!status approved by Director, AJPO 88-02-05
!status approved by WG9/Ada Board 87-12-07
!status approved by Ada Board 87-07-30
!status panel/committee-approved 87-05-06 (reviewed)
!status panel/committee-approved (9-0-0) 87-02-17 (pending editorial review)
!status work-item 87-01-19
!status received 86-06-19
!references 83-00746
!topic Predefined logical operators for boolean arrays

!summary 87-01-21

Predefined logical operations on boolean arrays are performed on a component-by-component basis, using the predefined logical operation for the component type (even if a user-defined logical operation for the component type is visible and hides the predefined one).

!question 87-12-07

Consider the following example:

```
procedure EXAMPLE is
    package P is
        type NB is new BOOLEAN;                              -- (1)
        function "and" (LEFT, RIGHT : in NB) return NB;      -- (2)
    end P;
    package body P is
        function "and" (LEFT, RIGHT : in NB) return NB is
        begin
            return NB(not (BOOLEAN(LEFT) and BOOLEAN(RIGHT)));
        end "and";
    end P;
begin
    declare
        use P;
        type ARR is array (INTEGER range <>) of NB;          -- (3)
        A1, A2, B : ARR (1..4);
    begin
        A1 := (TRUE, TRUE, FALSE, FALSE);
        A2 := (TRUE, FALSE, TRUE, FALSE);
        B := A1 and A2;                                      -- (4)
    end;
end EXAMPLE;
```

The predefined logical operators for the boolean type NB are implicitly declared at (1). The predefined AND operator for NB is overloaded and hidden by the user-defined AND operator at (2). A one-dimensional array type whose components are of type NB is declared at (3), together with the predefined logical operators for this one-dimensional array type. What does the AND operator applied to A1 and A2 do? In particular, which AND operation for components is used at (4) (the hidden predefined one or the user-defined one)?

!response 87-01-21

The only AND operator applicable at (4) is the predefined operator implicitly declared at (3). 4.5.1(3) states:

> The operations on arrays are performed on a component-by-component basis on matching components, if any (as for equality, see 4.5.2). ...

The parenthetical phrase not only defines how components match, but also indicates which operation is applied on a component-by-component basis. For the equality operation on arrays, 4.5.2(5) states:

> For two array values ... of the same type, the left operand is equal to the right operand if and only if for each component of the left operand there is a matching component of the right operand and vice versa; and the values of matching components are equal, as given by the predefined equality operator for the component type.

Thus, logical operations on boolean arrays are performed on matching components, using the corresponding predefined logical operation for the component type (even if a user-defined logical operator for the same component type is visible and hides the predefined one). Thus, the AND operation that is applied on a component-by-component basis at (4) is the hidden predefined operation implicitly declared at (1), not the user-defined operation declared at (2).

At (4), B should be assigned the value (TRUE, FALSE, FALSE, FALSE), not the value (FALSE, TRUE, TRUE, TRUE).

!standard 09.04 (06) 88-05-23 AI-00441/06
!class ramification 86-07-10
!status approved by WG9/AJPO 88-02-05
!status approved by Director, AJPO 88-02-05
!status approved by WG9/Ada Board 87-12-07
!status approved by Ada Board 87-07-30
!status panel/committee-approved (10-0-3) 87-02-17 (by ballot)
!status panel/committee-approved (8-1-1) 86-11-13 (pending letter ballot)
!status work-item 86-08-12
!status received 86-07-10
!references 83-00775
!topic A task without dependents can be completed but not terminated

!summary 87-01-19

A task that has no dependent tasks can be completed but not yet terminated, i.e., T'CALLABLE can be FALSE when T'TERMINATED is not yet TRUE.

!question 86-12-18

9.4(6) says:

> If a task has no dependent task, its termination takes place when it has completed its execution. ... If a task has dependent tasks, its termination takes place when the execution of the task is completed and all dependent tasks are term- inated.

For a task with no dependents, does the use of "when" imply that termination takes place at the same time as completion? In particular, if a task without dependents is aborted, is it possible for T'CALLABLE to be FALSE and T' TERMINATED also to be FALSE?

!response 87-12-07

In 9.4(6), "when" specifies a condition that is to be satisfied, not a time at which an action occurs. In particular, since completion and termination are distinct states of a task (as is indicated by the rule for tasks that have dependents), there is no reason to read the rules as specifying that the transition between these states takes place "instantaneously" when the specified conditions are satisfied. In general, a task need not be terminated as soon as the conditions for termination are satisfied. In short, it is possible for a task without dependents to be completed but not yet terminated, i.e., T'TERMINATED can be FALSE even though T'CALLABLE is FALSE. This is especially the case if a task is aborted while engaged in a rendezvous with the caller.

!standard 11.04.01 (03) 88-06-29 AI-00455/05
!class binding interpretation 86-08-12
!status approved by WG9/AJPO 88-02-05
!status approved by Director, AJPO 88-02-05
!status approved by WG9/Ada Board 87-12-07
!status approved by Ada Board (21-0-0) 87-02-19
!status panel/committee-approved 86-10-15 (reviewed)
!status panel/committee-approved (5-0-0) 86-09-11 (pending editorial review)
!status work-item 86-08-12
!status received 86-08-12
!references 83-00772
!topic Raising an exception before the sequence of statements

!summary 86-09-17

If an exception is raised due to the attempt to activate a task and the exception is raised after the elaboration of a declarative part and just before the execution of a sequence of statements, the sequence of statements is not executed and control is transferred in the same manner as for an exception raised in the sequence of statements.

!question 88-06-29

11.4.1(1-4) specify the effect of raising an exception "in" the sequence of statements. 9.3(2-3) explain that the activation of a task can occur just before the sequence of statements in a frame, and that this activation attempt can cause an exception to be raised (PROGRAM_ERROR or TASKING_ERROR) if the attempt is unsuccessful. Since the raising of such exceptions is not, strictly speaking, covered by the wording of 11.4.1(1-4), is not the effect of raising them undefined?

!recommendation 86-09-17

If an exception is raised due to the attempt to activate a task and the exception is raised after the elaboration of a declarative part and just before the execution of a sequence of statements, the sequence of statements is not executed and control is transferred in the same manner as for an exception raised in the sequence of statements.

!discussion 86-09-17

The discussion of the effect of raising an exception in 11.4.1 does not cover the case when the exception is raised by the attempt to activate a task object declared by an object declaration, since this attempt occurs just before the execution of a sequence of statements and just after completing the elaboration of a declarative part. Although the wording does not, technically speaking, cover this case, the intent is clear: raising such an exception means the sequence of statements is not executed and control is transferred in the same manner as for an exception raised in the sequence of statements.

Ada Commentary ai-00464-bi.wj downloaded on Wed Aug 10 09:59:55 EDT 1988

```
| !standard 09.06    (01)                              88-05-23  AI-00464/05
  !class binding interpretation 86-10-02
| !status approved by WG9/AJPO 88-02-05
| !status approved by Director, AJPO 88-02-05
| !status approved by WG9/Ada Board 87-12-07
  !status approved by Ada Board 87-07-30
  !status panel/committee-approved 87-02-18 (reviewed)
  !status panel/committee-approved (8-0-0) 86-11-13 (pending editorial review)
  !status work-item 86-10-15
  !status received 86-10-02
  !references AI-00222, 83-00813
  !topic Delay statements executed by the environment task
```

!summary 87-01-19

Delay statements can be executed by the environment task when a library package is elaborated. Such statements delay the environment task.

!question 86-10-16

9.6(1) says:

> The execution of a delay statement ... suspends further execution of the task that executes the delay statement ...

If a delay statement is given in the statements of a library package or in a subprogram that is executed during the elaboration of a library package, what task is delayed?

!recommendation 87-01-19

Delay statements can be executed by the environment task when a library package is elaborated. Such statements delay the environment task.

!discussion 87-01-19

10.1(8) says:

> Each main program acts as if called by some environment task;

For this reason, a delay statement is allowed in the main program and delays the environment task. Since library packages are elaborated by the environment task (see AI-00222), delay statements are also allowed in library packages. Delay statements executed by the environment task during the elaboration of library units delay the environment task.

Ada Commentary ai-00466-bi.wj downloaded on Wed Aug 10 09:59:54 EDT 1988

| !standard 14.01 (07) 88-05-23 AI-00466/04
!class binding interpretation 86-10-02
| !status approved by WG9/AJPO 88-02-05
| !status approved by Director, AJPO 88-02-05
| !status approved by WG9/Ada Board 87-12-07
!status approved by Ada Board 87-07-30
!status panel/committee-approved 87-01-19 (reviewed)
!status panel/committee-approved (8-0-0) 86-11-13 (pending editorial review)
!status work-item 86-10-15
!status received 86-10-02
!references AI-00222, AI-00399, 83-00814
!topic I/O performed by library tasks

!summary 86-12-15

The language does define what happens to external files after the completion of the main program and before completion of all the library tasks.

!question 86-10-16

14.1(7) says:

> The language does not define what happens to external files after the completion of the main program (in particular, if corresponding files have not been closed).

Suppose an external file is opened by a task whose master is a library package and execution of the main program is completed before the library task terminates. Does 14.1(7) mean to imply that the external file can be closed implicitly after completion of the main program even though the file is still being accessed by the library task?

!recommendation 86-12-15

The language does define what happens to external files after the completion of the main program and before completion of all the library tasks.

!discussion 86-12-15

AI-00399 considers the program as a whole terminated when the main program and all library tasks have terminated. 14.1(7) was only intended to note that when the main program and all library tasks have finished, the disposition of external files is not further specified by the language. In particular, whether the files are closed, saved, or modified, depends on the operating system and actions by operators, programmers, etc. However, the intent was that any I/O operations performed by library tasks be executed in accordance with the Standard even if the main program has completed (e.g., even if the main program has a null body).

Ada Commentary ai-00467-cr.wj downloaded on Tue Aug 9 14:29:47 EDT 1988

```
| !standard 03.05.10 (08)                              88-05-23  AI-00467/04
  !class correction 86-10-10
| !status approved by WG9/AJPO 88-02-05
| !status approved by Director, AJPO 88-02-05
| !status approved by WG9/Ada Board 87-12-07
| !status approved by Ada Board 87-07-30
  !status panel/committee-approved 87-01-19 (reviewed)
  !status panel/committee-approved (5-0-1) 86-11-14 (pending editorial review)
  !status work-item 86-10-10
  !status received 86-10-02
  !references 83-00803
  !topic Correction to AI-00179/06
```

!summary 86-10-13

In the discussion section of AI-00179/06, the upper bound of SG's range constraint and the model interval in the subsequent discussion are incorrect because the model numbers for subtype SG are the same as the model numbers for type G.

!question 86-10-10

The discussion section of AI-00179/06 contains the following example:

```
type G is delta 0.01 range 1.00 .. 10.00;
for G'SMALL use 0.01;
subtype SG is F delta 0.1 range 1.0 .. 9.95;
```

The discussion then goes on to say that for subtype SG, 9.9 and 10.0 are consecutive model numbers. This statement is incorrect since a length clause specifying SMALL is given for type G, and hence, SG and G have the same model numbers.

The example should be corrected so SG has the upper bound 9.995, and the following discussion should be modified accordingly.

!recommendation 86-10-13

Correct the discussion of AI-00179 by replacing:

```
|           type G is delta 0.01 range 1.00 .. 10.00;
            for G'SMALL use 0.01;
            subtype SG is F delta 0.1 range 1.0 .. 9.95;
```

For the subtype SG, 9.9 and 10.0 are consecutive model numbers (3.5.9(14)). It is implementation dependent whether the upper bound of SG is represented as the model number 9.9 or the model number 10.0. Depending on the implementation's choice, the value returned by SG'FORE will be either 2 or 3. In addition, note that the bounds of SG need not be given by static expressions. If the upper bound is non-static and has a value lying in the model interval 9.9 to 10.0, SG'FORE's value will be implementation dependent (and must be computed at run-time). The fact that 'FORE may return implementation dependent values should be taken into consideration by programmers. |

with:

```
type G is delta 0.01 range 1.00 .. 10.00;
for G'SMALL use 0.01;
subtype SG is F delta 0.01 range 1.00 .. 9.995;
```

For the subtype SG, 9.99 and 10.00 are consecutive model numbers (3.5.9(14)). It is implementation dependent whether the upper bound of SG is represented as the model number 9.99 or the model number 10.0. Depending on the implementation's choice, the value returned by SG'FORE will be either 2 or 3. In addition, note that the bounds of SG need not be given by static expressions. If the upper bound is non-static and has a value lying in the model interval 9.99 to 10.00, SG'FORE's value will be implementation dependent (and must be computed at run-time). The fact that 'FORE may return implementation dependent values should be taken into consideration by programmers.

!discussion 86-10-10

For the discussion section of commentary AI-00179/06 (as approved by the AJPO, Ada Board, and ISO WG9) to be correct, the example should be changed so the upper bound of SG is 9.995. The model interval in the discussion will then range from 9.99 to 10.00.

Ada Commentary ai-00475-bi.wj downloaded on Wed Aug 10 09:59:25 EDT 1988

!standard 04.05.05 (08) 88-05-23 AI-00475/05
!class binding interpretation 86-10-13
!status approved by WG9/AJPO 88-02-05
!status approved by Director, AJPO 88-02-05
!status approved by WG9/Ada Board 87-12-07
!status approved by Ada Board 87-07-30
!status panel/committee-approved 87-05-06 (reviewed)
!status panel/committee-approved (8-0-0) 87-02-18 (pending editorial review)
!status work-item 87-01-15
!status received 86-10-13
!references 83-00829
!topic Multiplication of fixed point values by negative integers

!summary 87-03-17

If the integer in an integer multiplication of a fixed point value is negative, the multiplication is equivalent to changing the sign of the fixed point value followed by repeated addition.

!question 87-01-15

4.5.5(8) says:

> Integer multiplication of fixed point values is equivalent to repeated addition.

What is the effect of multiplication by a negative integer?

!recommendation 87-03-17

If the integer in an integer multiplication of a fixed point value is negative, the multiplication is equivalent to changing the sign of the fixed point value followed by repeated addition.

!discussion 87-01-15

Clearly the intent in defining the effect of multiplication of a fixed point value by an integer was to specify that the result have the proper sign.

```
!standard 06.03.01 (04)                                  88-05-23  AI-00493/05
!class ramification 86-11-10
!status approved by WG9/AJPO 88-02-05
!status approved by Director, AJPO 88-02-05
!status approved by WG9/Ada Board 87-12-07
!status approved by Ada Board 87-07-30
!status panel/committee-approved 87-05-06 (reviewed)
!status panel/committee-approved (9-0-0) 87-02-17 (pending editorial review)
!status work-item 87-01-16
!status received 86-11-10
!references 83-00840
!topic Operator symbols that represent the same operator
```

!summary 87-01-16

Two string literals serving as operator symbols represent the same operator if the string literals are identical or if the only difference is that some letters appear in upper case rather than lower case.

!question 87-03-16

6.3.1(4) says:

> A string literal given as an operator symbol can be replaced by a different string literal if and only if both represent the same operator.

What is meant by "represent the same operator"? In particular, consider the following example:

```
function "-" (X, Y : INTEGER) return INTEGER renames STANDARD."+";

procedure P (X : INTEGER := "+"(3, 4));

procedure P (X : INTEGER := "-"(3, 4)) is       -- legal? (no)
begin ... end P;
```

Do the operator symbols "+" and "-" represent the same operator?

!response 87-01-16

6.1(3) says "the case of letters [in an operator symbol] is not significant." This means the string literals "AND" and "and", for example, can serve as operator symbols for the same operator. This is the sense in which different operator symbols can represent the same operator.

6.3.1(5) requires that corresponding lexical elements be given the same meaning by the visibility and overloading rules, i.e., corresponding lexical elements must be associated with the same declaration (8.3(2-3)). In the example given, the two subprogram specifications do not conform because the operator symbols are associated with different declarations by the visibility rules.

Ada Commentary ai-00502-cr.wj downloaded on Wed Aug 10 09:59:02 EDT 1988

!standard 08.05 (05) 88-05-23 AI-00502/05
!class correction 86-11-19
!status approved by WG9/AJPO 88-02-05
!status approved by Director, AJPO 88-02-05
!status approved by WG9/Ada Board 87-12-07
!status approved by Ada Board 87-07-30
!status panel/committee-approved 87-05-06 (reviewed)
!status panel/committee-approved (9-0-0) 87-02-17 (pending editorial review)
!status work-item 87-01-16
!status received 86-11-19
!references 83-00859
!topic Error in AI-00170/06

!summary 87-01-16

The declaration of type SINT in the question's example should be replaced with a subtype declaration so SINT has the type INTEGER.

!question 87-03-17

The following example appears in AI-00170/06:

```
        type SINT is range 0..100;
        type VREC (N : SINT := 0) is
             record
                  S : STRING (1..N);
             end record;
```

Isn't the component declaration illegal since STRING's index subtype is POSITIVE?

!recommendation 87-01-16

Replace the declaration of SINT as follows:

```
        subtype SINT is NATURAL range 0..100;
```

!discussion 87-03-17

Since STRING's index subtype is POSITIVE, SINT must be declared as a subtype of INTEGER.

Ada Commentary ai-00503-cr.wj downloaded on Wed Aug 10 09:59:01 EDT 1988

```
| !standard 13.07.02 (07)                               88-05-23  AI-00503/04
  !standard A         (34)
  !class correction 86-11-19
| !status approved by WG9/AJPO 88-02-05
| !status approved by Director, AJPO 88-02-05
| !status approved by WG9/Ada Board 87-12-07
  !status approved by Ada Board 87-07-30
  !status panel/committee-approved (9-0-0) 87-02-17
  !status work-item 87-01-16
  !status received 86-11-19
  !references 83-00860
  !topic Error in AI-00258/05
```

!summary 87-01-16

The renaming declaration for Ren_C2 in AI-00258/05 is illegal since a type mark is required instead of a subtype indication. An appropriate subtype declaration should be added to the example.

!question 87-01-16

AI-00258/05 contains the following declarations:

```
package P is
     Obj    : R;
     Ren_C2 : String (1..N) renames F.all.C2;
end P;
```

Since a renaming declaration requires a type mark instead of a subtype indication (see 8.5(2)), isn't the declaration of Ren_C2 illegal?

!recommendation 87-01-16

Replace the declaration of package P with the following:

```
package P is
     subtype STR_N is STRING(1..N);
     Obj    : R;
     Ren_C2 : STR_N renames F.all.C2;
end P;
```

!discussion 87-01-16

Since a renaming declaration requires a type mark instead of a subtype indication, STRING(1..N) must be replaced with the name of an appropriate subtype.

Ada Commentary ai-00508-bi.wj downloaded on Wed Aug 10 09:58:39 EDT 1988

!standard 03.05.09 (11) 88-05-23 AI-00508/03
!class binding interpretation 86-11-14
!status approved by WG9/AJPO 88-02-05
!status approved by Director, AJPO 88-02-05
!status approved by WG9/Ada Board 87-12-07
!status approved by Ada Board 87-07-30
!status panel/committee-approved 87-03-16 (reviewed)
!status panel/committee-approved (6-0-0) 86-11-14 (pending editorial review)
!status work-item 86-08-08
!status received 84-03-13
!references 83-00750
!topic The safe numbers of a fixed point subtype

!summary 86-12-28

The safe numbers of a fixed point subtype are the safe numbers of its base type.

!question 86-12-28

3.5.9(11) defines the safe numbers of a fixed point TYPE. Shouldn't there also be a definition for the safe numbers of a fixed point SUBtype (as there is for floating point subtypes; see 3.5.7(9)).

!recommendation 86-12-28

The safe numbers of a fixed point subtype are the safe numbers of its base type.

!discussion 87-03-13

Since 3.5.7(9) defines the safe numbers of a floating point subtype, a similar definition should have been given for fixed point subtypes. The intended definition was that the safe numbers of a subtype be those of its base type. Safe numbers are used (in the Standard) to specify the accuracy of real numeric operations. 4.5.7(8) defines the accuracy of all real numeric operations in terms of the safe numbers (since the model numbers of a type are a subset of the safe numbers and the rules for computing with safe numbers are the same as the rules for computing with model numbers). Since all numeric operations are declared for types rather than subtypes, these operations are performed using the safe numbers of the base type, and hence, it was intended for the safe numbers of a real subtype to be the same as the safe numbers of its base type.

Ada Commentary ai-00511-cr.wj downloaded on Wed Aug 10 09:58:37 EDT 1988

```
!standard 02.08    (04)                              88-05-23  AI-00511/05
!class correction 87-01-13
!status approved by WG9/AJPO 88-02-05
!status approved by Director, AJPO 88-02-05
!status approved by WG9/Ada Board 87-12-07
!status approved by Ada Board 87-07-30
!status panel/committee-approved 87-05-06 (reviewed)
!status panel/committee-approved (9-0-0) 87-02-17 (pending editorial review)
!status work-item 87-01-16
!status received 87-01-13
!references 83-00871
!topic Error in AI-00388/04
```

!summary 87-01-16

The example given in the response is syntactically incorrect unless "package P" is replaced either with "procedure P" or with "package P is end".

!question 87-01-16

The response section contains the following example, which is asserted to be legal:

```
generic
     pragma PAGE;
     X : INTEGER;
     pragma PAGE;
package P;
```

Isn't the example illegal because "is end" is required after "package P"?

!recommendation 87-01-13

Rewrite the example in the response section as follows:

```
generic
     pragma PAGE;
     X : INTEGER;
     pragma PAGE;
procedure P;
```

!discussion 87-03-17

To make the example syntactically legal, "package" should be replaced with "procedure".

Ada Commentary ai-00516-bi.wj downloaded on Tue Aug 9 17:00:46 EDT 1988

| !standard 04.05.07 (04) 88-05-23 AI-00516/05
!class binding interpretation 86-10-13
| !status approved by WG9/AJPO 88-02-05
| !status approved by Director, AJPO 88-02-05
| !status approved by WG9/Ada Board 87-12-07
!status approved by Ada Board 87-07-30
!status panel/committee-approved 87-05-06 (reviewed)
!status panel/committee-approved (8-0-0) 87-02-18 (pending editorial review)
!status work-item 87-01-15
!status received 86-10-13
!references 83-00829
!topic The safe interval for a fixed/integer result

!summary 87-03-18

When a fixed point value is divided by an integer value, the result model interval is determined by considering the integer value to be a model interval consisting of a single integer value.

!question 87-03-18

4.5.5(8) says:

> Division of a fixed point value by an integer does not involve a change in type but is approximate (see 4.5.7).

The rules in 4.5.7 do not, however, specify what the model interval of an integer is, so, strictly speaking, the accuracy of such divisions is not defined. Is the intent to apply the usual rules given that the integer value is a safe number?

!recommendation 87-03-18

For purposes of applying the rules in 4.5.7, an integer value is considered to be a model interval consisting of a single integer value.

!discussion 87-06-11

Clearly, when determining the result model interval for division of a fixed point value by an integer, the rules in 4.5.7 should be applied with the integer value being considered to be a model interval consisting of a single value.